
# Benchmarking von Hochleistungsgrafikkarten

**eingereicht als**

**DIPLOMARBEIT**

**an der**

**HOCHSCHULE MITTWEIDA**

**UNIVERSITY OF APPLIED SCIENCES**

**Mathematik / Naturwissenschaften / Informatik**

**Manching und Mittweida, 2009**

Erstprüfer: Prof. Dr.-Ing. Olaf Hagenbruch (Fakultät IT & ET)

Zweitprüfer: Burkhard Balser (Manager of New Avionicstructures, EADS)

Dipl.-Ing. (BA) Florian Biechele (EADS)

vorgelegt von: Ronny Lauenstein (WF03)

ronny.lauenstein@web.de

# Inhaltsverzeichnis

## Abkürzungsverzeichnis

| | |
|---|---|
| AGP | Accelerated Graphics Port |
| AMD | Advanced Micro Devices |
| API | Advanced Programming Interface |
| CAL | Compute Abstraction Layer |
| Cg | C for graphics |
| CGA | Color Graphics Adapter |
| COTS | Commercial Off-the-Shelf |
| cPCI | CompactPCI |
| CPU | Central Processing Unit |
| CTA | Cooperative Thread Array |
| CTM | Close to Metal |
| CUDA | Compute Unified Device Architecture |
| DVI | Digital Visual Interface |
| ECC RAM | Error Correction Code RAM |
| EDO RAM | Extended Data Output RAM |
| EGA | Enhanced Graphics Adapter |
| flops | Floating-Point Operations per Second |
| FPM DRAM | Fast Page Mode Dynamic RAM |
| fps | Frames pro Sekunde |
| FPU | Floating-point unit |
| Gb | Gigabit |
| GB | Gigabyte |
| GDDR RAM | Graphics Double Data Rate RAM |
| Gflops | Millionen Floating-Point Operations per Second |
| GPGPU | General Purpose Computation on Graphics Processing Unit |
| GPU | Graphics Processing Unit |
| HDCP | High-bandwidth Digital Content Protection |
| HDMI | High Definition Multimedia Interface |
| HDSDI | High Definition Serial Digital Interface |
| HPC | High Performance Computing |
| IBM | International Business Machines Corporation |
| IGP | Integrated Graphics Processor |
| Mb | Megabit |
| MDA | Monochrome Display Adapter |

| | |
|---|---|
| MIMD | Multiple Instruction Multiple Data |
| MOTS | Modified Off-the-Shelf |
| MRT | Magnet Resonanz Tomografie |
| OpenCL | Open Computing Language |
| OpenGL | Open Graphics Library |
| PAE | Physical Address Extension |
| PC | Personal Computer |
| PCI | Peripheral Component Interconnect |
| PCIe | PCI-Express |
| PTX | Parallel Thread Execution |
| RAM | Random Access Memory |
| RAM-DAC | Random Access Memory - Digital/Analog Converter |
| RGB | Rot, Grün, Blau |
| ROTS | Ruggedized Off-the-Shelf |
| SDK | Software Development Kit |
| SFU | Special Function Units |
| SGI | Silicon Graphics |
| SIMD | Single Instruction Multiple Data |
| SiS | Silicon Integrated Systems |
| SLI | Scalable Link Interface |
| SM | Streaming Multiprozessor |
| SP | Stream Prozessor |
| SPL | System Prototyping Labor |
| TMDS | Transition Minimized Differential Signaling |
| TPC | Texture Prozessor Cluster |
| TRL | Technology Readiness Level |
| UDI | Unified Display Interface |
| VGA | Video Graphics Array |
| VME | Versa Module Eurocard |
| VRAM | Video RAM |

# Abbildungsverzeichnis

# Tabellenverzeichnis

## Listings

# 1 Einleitung

## 1.1 Problemstellung

Themensteller der vorliegenden Diplomarbeit ist die Abteilung MEA15 – New Avionicstructures des Bereiches Military Air Systems der Firma EADS Defence & Security mit Sitz in Manching.

MEA15 ist zuständig für die Entwicklung und den Test von neuen Avioniksystemen und Konzepten bis *Technology Readiness Level 6 (TRL)*. Die entwickelten Komponenten werden im angeschlossenen *System Prototyping Labor (SPL)* prototypenhaft implementiert und mit Hilfe von hochleistungsfähigen Rechneranlagen oder im Flug getestet.

Im Bereich militärischer Flugzeugsysteme besteht das Problem Trefferwahrscheinlichkeiten zu berechnen und eigene Kollisionen zu vermeiden. Hierfür gibt es von Fremdfirmen entwickelte, flugfähige Spezialentwicklungen. Diese weisen jedoch ein unzureichendes Leistungsniveau auf.

Um dieses Problem zu beheben, bestehen Bestrebungen ein eigenes Jet-internes System zu entwickeln, welches die Kollisionswahrscheinlichkeit zwischen Objekten in Echtzeit aufzeigen kann. Eine Idee liegt hierbei in der Implementierung eines *Künstlichen Neuronalen Netzes*, welches sich in Zukunft auf vorhandenen flugfähigen Standard-Komponenten betreiben lässt. Zum Einsatz kommen hierbei speziell gehärtete *Versa Module Eurocard (VME)*- oder *CompactPCI (cPCI)*-Systeme. cPCI-Hardware ist unter anderem auch in der industriellen Steuerungstechnik vorzufinden.

In der Abteilung MEA1 existiert bereits ein Neuronales Netz. Dieses ist prinzipiell zur Berechnung von Kollisionswahrscheinlichkeiten geeignet. Problematisch ist jedoch, dass das Training des Neuronalen Netzes bisher viel zu lange dauert. Ein Grund dafür, dass das bisherige Neuronale Netz momentan nicht produktiv angewendet wird, liegt unter anderem in der zu langen Berechnungsdauer. Um die Trainingsdauer zu minimieren, könnten moderne Grafikkarten helfen.

Durch die enorme parallele Rechenleistung moderner Grafikkarten und der zahlreichen Recheneinheiten eines Grafikprozessors (bis zu 240 Cores) sollen anspruchsvolle mathematische Berechnungen in deutlich kürzerer Zeit als nur mithilfe der *Central Processing Unit (CPU)* durchführbar sein.

## 1.2 Zielsetzung

Bezugnehmend auf die zuvor erläuterte Problematik soll im Rahmen dieser Diplomarbeit ein Programm entwickelt werden, auf dessen Grundlage eine spätere Weiterentwicklung und Erweiterung für andere Neuronale Netzprobleme möglich ist. Durch die anschließende Portierung des Neuronalen Netzes auf die *Graphics Processing Unit (GPU)* einer zuvor ausgewählten Grafikkarte soll untersucht werden, ob die bisherige Berechnungsdauer signifikant verringert werden kann. Darüber hinaus soll die vorliegende Arbeit zu einem besseren (firmeninternen) Verständnis rund um das Thema Grafikkarten und Neuronale Netze beitragen.

Um dieses Ziel zu erreichen, sollen zunächst die technischen und theoretischen Grundlagen Neuronaler Netze und Grafikkarten erörtert werden. In diesem Zusammenhang soll auf die Funktionsweise einer Grafikkarte näher eingegangen werden.

Des Weiteren soll die Arbeit eine Hilfestellung geben, welche Grafikhardware für die mathematischen Berechnungen geeignet ist und in Zukunft angeschafft werden könnte (Hardwareeignung). Zu diesem Zweck soll ein aktueller Marktüberblick über handelsübliche Grafikkarten im PC-Umfeld sowie Grafikboards im VME- und cPCI-Bereich mit entsprechender Spezifikation erstellt werden. Darauf aufbauend sollen die aktuell am Markt erhältlichen Grafikkarten und Grafikboards auf ihre Eignung hin evaluiert werden, um so eine geeignete Grafikkarte für die mathematischen Berechnungen zu ermitteln.

In einem weiteren Schritt soll in Koordination mit der Fachabteilung ein Neuronales Netz zum Test für die Leistungsbewertung der ausgewählten Grafikkarte entworfen werden. Hierbei handelt es um eine spezifische Leistungsbewertung der Hardware für die Berechnung und das Training des Neuronalen Netzes.

Aufbauend auf diesem Entwurf wird in einem nächsten Schritt die Implementierung des Testprogramms vorgenommen. Um festzustellen, ob ein signifikanter Leistungs-unterschied vorliegt und um eine Messung der jeweiligen Berechnungsdauer vorzunehmen, soll das Neuronale Netz sowohl auf die CPU als auch auf die GPU portiert werden.

Im Anschluss an die Implementierung erfolgt eine Leistungsbewertung, bei der die Eignung der, auf Basis der Marktuntersuchung ausgewählten, Grafikkarte für die Berechnung des Neuronalen Netzes untersucht werden soll.

Kann durch die Verwendung einer handelsüblichen Grafikkarte die Berechnungsdauer des Neuronalen Netzes signifikant verringert werden, führt dies nicht nur zu einem

Effizienzmehrwert, sondern zugleich zu einem Effektivitätsmehrwert für den Bereich MEA1.

Die nachfolgende Abbildung 1-1 gibt einen zusammenfassenden Überblick über die zuvor dargestellte Vorgehensweise.

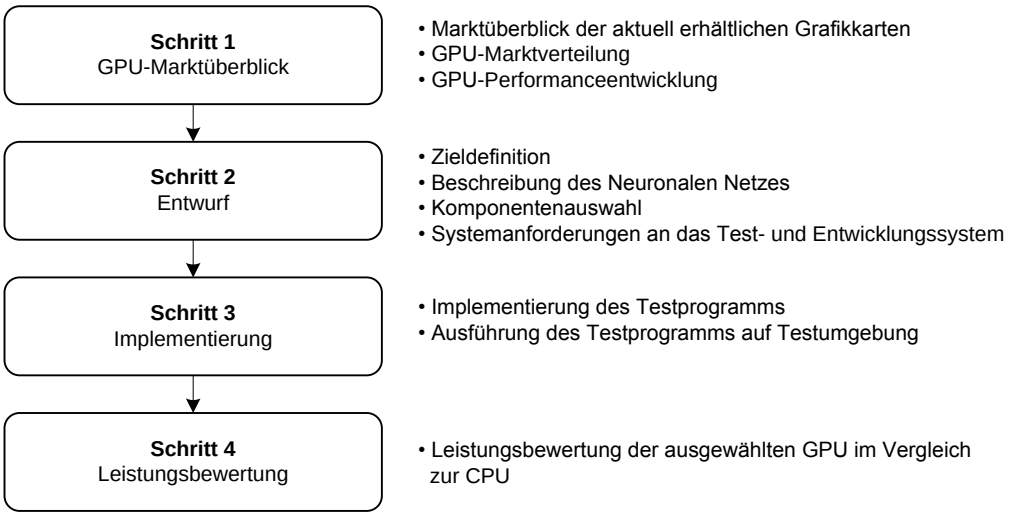



**Abbildung 1-1:** Vorgehensweise

## 1.3 Aufbau

Um das in Abschnitt 1.2 genannte Ziel zu erreichen, werden in Kapitel 2 zuerst die technischen und theoretischen Grundlagen aufbereitet. Hierzu werden in Abschnitt 2.1 Neuronale Netze beschrieben. Zunächst wird auf Biologische Neuronale Netze (Abschnitt 2.1.1) eingegangen. Anschließend werden die für die Arbeit relevanten Künstlichen Neuronalen Netze (Abschnitt 2.1.2) betrachtet. In Abschnitt 2.2 wird die allgemeine Funktionsweise einer Grafikkarte näher erläutert. Nachfolgend werden die Komponenten der Grafikkarte Grafikspeicher (Abschnitt 2.2.1), RAM-DAC (Abschnitt 2.2.2) und DVI sowie TMDS (Abschnitt 2.2.3) näher betrachtet. In Abschnitt 2.3 wird auf den Status Quo der GPU eingegangen. In diesem Zusammenhang wird die Marktverteilung (Abschnitt 2.3.1) aufgezeigt, ein Überblick über den Stand der Technik gegeben (Abschnitt 2.3.2) sowie die Performanceentwicklung (Abschnitt 2.3.3) untersucht. Im Anschluss wird in Abschnitt 2.4 das GPU-Programmiermodell aufbereitet. Nach einem kurzen Einblick in das Unified Shader-Modell (Abschnitt 2.4.1) wird darauf aufbauend die nVIDIA CUDA-Hardwarearchitektur (Abschnitt 2.4.2) erklärt. Abschließend wird ein Einblick in das nVIDIA CUDA-Programmiermodell (Abschnitt 2.4.3) gegeben.

In Kapitel 3 der Arbeit wird der Entwurf modelliert. Hierzu wird zunächst in Abschnitt 3.1 die an die konkreten Gegebenheiten angepasste Zieldefinition formuliert. Anschließend wird in Abschnitt 3.2 der Netzentwurf erarbeitet, der insbesondere die Parallelisierbarkeit von Neuronalen Netzen beinhaltet. Abschnitt 3.3 beschäftigt sich mit Überlegungen zur Komponentenauswahl. Hierbei werden die relevanten Systemanforderungen (Abschnitt 3.3.1), das Testsystem (Abschnitt 3.3.2) sowie die ersichtlichen Möglichkeiten, Grenzen und Probleme (Abschnitt 3.3.3) betrachtet.

Aufbauend auf den Entwurf erfolgt in Kapitel 4 die Implementierung des Testprogramms. Hierzu wird in Abschnitt 4.1 zunächst das Kollisionsfeld implementiert. Als nächstes erfolgt die Implementierung des Neuronalen Netzes in Abschnitt 4.2. Die Aspekte der Programmoptimierung werden in Abschnitt 4.3 angesprochen.

Im Anschluss an die Implementierung erfolgt in Kapitel 5 eine Leistungsbewertung des Programms. Abschnitt 5.1 beschreibt hierzu zunächst die Vorgehensweise der Leistungsbewertung. In Abschnitt 5.2 wird die Matrizenmultiplikation und in Abschnitt 5.3. die Berechnungsdauer des Neuronalen Netzes dargestellt.

Den Abschluss der Arbeit bildet Kapitel 6 mit einer Zusammenfassung der Ergebnisse (Abschnitt 6.1) sowie einem Ausblick auf die zukünftige Entwicklungen (Abschnitt 6.2).

# 2 Technische und theoretische Grundlagen

Im folgenden Kapitel sollen die technischen und theoretischen Grundlagen für die Arbeit aufbereitet werden. Hierzu wird in Abschnitt 2.1 zunächst auf Neuronale Netze eingegangen. Abschnitt 2.2 beschreibt die allgemeine Funktionsweise einer Grafikkarte. Anschließend wird in Abschnitt 2.3 auf den Status Quo der GPU eingegangen. Abschließend werden in Abschnitt 2.4 die Grundlagen der GPU-Programmierung erläutert.

## 2.1 Neuronale Netze

Nachfolgend wird zunächst auf das Biologische Neuronale Netz eingegangen, um darauf aufbauend die für die Arbeit relevanten Grundlagen des Künstlichen Neuronalen Netzes zu erläutern.

### 2.1.1 Biologische Neuronale Netze

Um Künstliche Neuronale Netze zu verstehen, ist es zunächst sinnvoll das Nervensystem des Menschen zu betrachten. Das menschliche Gehirn ist eine schwammige Masse von der Größe einer Pampelmuse. Gleichzeitig ist es mit seinen rund $10^{10}$ Nervenzellen ein Universum für sich [Gaud06, 4ff.]. Durch moderne medizinische Untersuchungsgeräte, wie z. B. der *Magnet Resonanz Tomografie (MRT)*, wurden erhebliche Fortschritte auf diesem Forschungsgebiet erzielt. Dennoch ist man noch nicht in der Lage die kognitiven Leistungen eines Menschen zu erklären.

Der Sitz der intelligenten Leistung befindet sich in der Hirnrinde (Neokortex), welche äußerlich etwa 0,5 m² groß ist. Diese zwei bis vier Millimeter dicke Schicht aus Nervenzellen (graue Materie) ist aus Platzgründen gefaltet. Das Innere (weiße Materie) enthält die Axonen (Nervenfasern) mit Myelinhüllen. In jedem Quadratmillimeter Hirnrinde befinden sich etwa 100.000 vernetzte Nervenzellen (Neuronen). An einem Neuron lassen sich drei Hauptstrukturen unterscheiden: Dendritenbaum, Zellkörper und Axon. In der Computerwelt würde man sagen: Eingabe – Verarbeitung – Ausgabe. Werden mehrere Neuronen zusammengeschaltet, so entsteht ein Neuronales Netz. Die nachfolgende Abbildung 2-1 (in Anlehnung an [Mind02]) zeigt den schematischen Aufbau einer Nervenzelle.

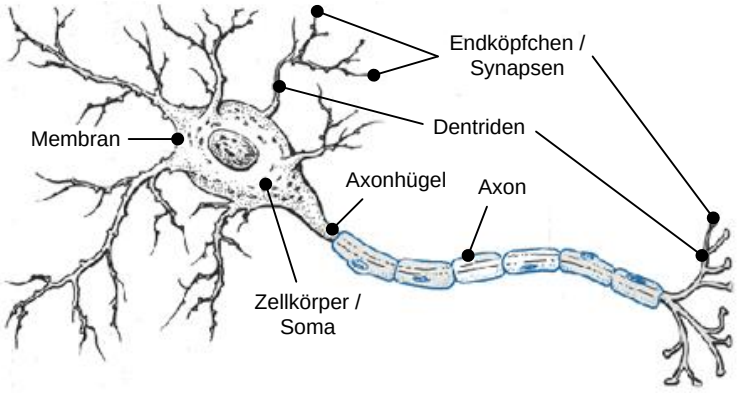



**Abbildung 2-1:** Biologischer Aufbau einer Nervenzelle [Mind02]

Bei der Geburt eines Menschen sind bereits alle Neuronen vorhanden. Neueste Untersuchungen gehen allerdings davon aus, dass Neuronen auch während des Lebens gebildet werden können. Es besteht eine Vorvernetzung auf Basis erblich genetischer Informationen. Die weitere Vernetzung wird durch Lernprozesse erzielt. Signale verändern dabei nicht das Innere eines Neurons, sondern die Verbindungen im Netzwerk. Ein einzelnes Neuron lässt keine Schlussfolgerung auf den eigentlichen Gedanken zu.

Vergleicht man das Gehirn (Neuron) mit einem Computer (Transistor), so wird man feststellen, dass ein Computer lediglich Binärzahlen speichert. Das menschliche Gehirn hingegen speichert Pattern (Muster). Alles was in einem Computer gespeichert oder verarbeitet werden soll, muss folglich zuvor in eine Zahl konvertiert werden. Ein Computer ist wahnsinnig schnell beim Berechnen und Sortieren von Zahlen – versagt jedoch bei Alltagsproblemen. In verschiedener Literatur findet man oftmals den Vergleich, dass Neuronen im Verhältnis zu einem modernen Logikgatter bis zu einer Millionen Mal langsamer arbeiten [AlBG07, 109; Urch02, 186]. Dieser Vergleich ist jedoch etwas unseriös, da die komplexe Informationsverarbeitung eines einzelnen Neurons nicht mit der einfachen Schaltfunktion eines Transistors vergleichbar ist.

Für die Übertragung eines Neuronalen Netzes auf den Computer gibt es Modellansätze, welche bis in die 1940er Jahre zurück gehen. Im Folgenden werden daher nun die für die Arbeit relevanten Künstlichen Neuronalen Netze betrachtet.

### 2.1.2 Künstliche Neuronale Netze

Bereits im Jahr 1943 schlugen McCulloch und Pitts ein „logisches Stellwertelement“ als Neuron vor [McPi43]. Jedoch ist das „Lernen“ beim McCulloch-Pitts-Neuron weder direkt erkennbar noch sind solche Netzwerke fehlertolerant. Ein fehlerfreies Arbeiten sämtlicher Komponenten ist zwingend erforderlich.

Der Psychologe Donald Hebb machte im Jahr 1949 einen Vorschlag wie Neuronale Netze lernen können. Nach der Hebbschen Lernregel werden Paare von Neuronen, die gleichzeitig aktiv sind, durch Veränderung der synaptischen Verbindungen stärker aneinander gebunden. Demzufolge wird die Verbindung zwischen zwei Neuronen verstärkt, wenn beide Neuronen zur gleichen Zeit aktiv sind [Hebb02]. Die Hebbsche Lernregel ist noch Bestandteil vieler Netzwerkmodelle, auch wenn die experimentelle Verifikation bis heute umstritten ist.

Ende der 1950er Jahre bot sich mit dem Aufkommen des Computers die Möglichkeit, die Lernfähigkeit von Neuronalen Netzen zu erproben und Anwendungsmöglichkeiten zu testen. Im Jahr 1958 stellte Frank Rosenblatt das *Perzeptron* als erstes lernfähiges Modell zur Klassifikation von visuellen Mustern anhand bekannter Klassifikations-beispiele vor [Rose58]. Das Rosenblatt-Perzeptron findet heute unter anderem noch Anwendung in der Beurteilung von Tumoren auf Röntgenaufnahmen. Heutzutage verbindet man mit dem Begriff Perzeptron ein einstufiges, lernfähiges Netz (single layer neuronal network).

Im Jahr 1960 entwickelten Bernard Widrow und Marcian Hoff das ADALINE-Netz (ADAptive LInear NEuron), das aus einem 3-Schichten-Netz mit Fehlerkorrektur besteht [Wild60]. Es kann beispielsweise als adaptiver Filter zur Verminderung von Übertragungsfehlern im Modem benutzt werden.

Im Jahr 1969 wiesen Minsky und Papert nach, dass die logische Funktionen XOR sich nicht mit einem Perzeptron erlernen lässt [MiPa69]. Dies war eine herbe Ernüchterung und ein gewaltiger Rückschlag, da es schon damals für solche Funktionen viel bessere und mathematisch einfachere Darstellungen gab.

Das sogenannte Multilayer-Perzeptron oder auch Backpropagation-Netzwerk wurde im Jahr 1974 von Paul Werbos vorgestellt. Das Multilayer-Perzeptron besteht aus einer Eingangsschicht, einer Ausgangsschicht und einer oder mehreren Zwischenschichten.

Im Jahr 1982 stellte der Physiker John Hopfield das Hopfield-Netz vor, das Rückkopplungen in das Modell von McCulloch und Pitts einfügt. Das Hopfield-Netz muss wie das Perzeptron in der Lernphase supervidiert (überwacht) werden.

Ebenfalls im Jahr 1982 stellte Teuvo Kohonen ein Modell selbstorganisierender Karten vor. Das sogenannte Kohonen-Modell oder Self Organizing Feature Map ist. Aufgrund seiner biologischen Orientierung und der einfachen Struktur ein in der Praxis oft verwendetes Modell.

Das Jahr 1986 führte zu einer Renaissance der Künstlichen Neuronalen Netze, als David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams den Backpropagation-Algorithmus wiederentdeckten [RuHW86]. Durch die zwischenzeitlich erheblich

gestiegene und preisgünstigere Rechenkapazität gegenüber den 1970er Jahren war es nun möglich, diesen Algorithmus erfolgreich auf eine Vielzahl sehr unterschiedlicher Problemstellungen anzuwenden.

Wie aus dem kurzen geschichtlichem Auszug zu erkennen ist, gibt es eine Vielzahl unterschiedlicher Modellansätze für Neuronale Netze, welche sich durch unterschiedliche Netzstrukturen und Lernverfahren unterscheiden. Diese werden nachfolgend erläutert.

**Netzstrukturen**

Die Struktur Neuronaler Netze kann vorwärtsgerichtet (Feedforward-Netze) oder rückgekoppelt (Feedback-Netze) sein. Bei *vorwärtsgerichteten Netzen* wird die Information ausgehend von der Eingangsschicht durch die verdeckten Schichten zur Ausgabeschicht verbreitet. Bei *rückgekoppelten Netzen*, wie dem Hopfield-Netz, hingegen werden die Ausgangsinformationen von Neuronen auf eine vorherige Schicht rückgekoppelt.

**Lernverfahren**

Die Lernverfahren Neuronaler Netze können in überwachtes, selbstorganisierendes oder nichtüberwachtes, stochastisches Lernen und Reinforcement Learning unterschieden werden.

Beim *überwachten Lernverfahren* werden Eingangs- und dazugehörige Ausgangsmuster vorgegeben. Das Netz berechnet anschließend ein Ausgangsmuster zu dem gegebenen Eingangsmuster und überprüft ob der Ausgabewert mit dem gegebenen Ausgangsmuster übereinstimmt. Weichen diese Werte voneinander ab, wird der Fehler berechnet und korrigiert. Hierzu werden die Gewichtungsfaktoren durch ein Lerngesetz solange angepasst bis das gewünschte Ausgangsmuster erreicht wird.

Wird nur das Eingangsmuster vorgegeben, bezeichnet man dies als *selbstorganisierendes Lernen*. Hierbei organisiert das Netz seine Reaktion auf das Eingangsmuster selbst. Das Neuron, das am stärksten durch ein Eingangsmuster „erregt“ wird, erhält eine Gewichtungsänderung. In dieser Lernphase bilden die einzelnen Neuronen somit selbständig bestimmte Musterklassen.

Unter *stochastischen Lernverfahren* werden Lernverfahren mit einem Zufallsanteil verstanden. Hierbei sucht das Netz selbständig nach Lösungen.

Beim *Reinforcement Learning* wird nach jeder Aktion eine Bewertung der Aktion errechnet. Das Netz verstärkt daraufhin die Gewichtungsfaktoren für gute Aktionen und schwächt diese für schlechte Aktionen.

Die Lernzeit ist die rechenaufwendigste Phase. Diese muss für gewöhnlich jedoch nur einmal durchgeführt werden. Danach erfolgt die praktische Anwendung des Neuronalen Netzes. Nachfolgend wird nun auf die Funktionsweise des Perzeptrons, eines einstufigen, lernfähigen Netzes (single layer neuronal network) mit überwachtem Lernen, eingegangen.

**Funktionsweise des Perzeptrons (single layer neuronal network)**

Das Perzeptron besteht aus einem Neuron, dem eine bestimmte Anzahl von Eingabemustern zugeführt wird.

Während der Lernphase werden die Eingabemuster $x_i$ und die entsprechenden Ausgabewerte *y* mit einer Zielgröße verglichen. Anschließend passt jedes Element seine Gewichtung *w* so an, dass es nur auf die Eingabemuster seiner Klasse reagiert. Hierfür benötigt man eine Lernregel, die angibt wie die Veränderungen vorgenommen werden sollen. Das Wissen eines Neuronalen Netzes ist also allein in seinen Gewichtungen gespeichert. Der Algorithmus des klassischen Rosenblatt-Perzeptron ist in [Shal08] detaillierter beschrieben. Zu Illustrationszwecken ist dieser Algorithmus als Funktion im Quellcode des Testprogramms vorhanden (siehe Anhang C).

Umgekehrt funktioniert das trainierte Perzeptron wie folgt. Der gewichtete Input wird Summiert und sobald das Neuron einen Schwellwert überschreitet wird es aktiv. Hierfür ist die Aktivierungsfunktion, teils auch Transferfunktion (Threshold) genannt, zuständig. Der Schwellwert kann zusätzlich durch eine Schwelle, den Bias *b*, beeinflusst werden. Der Bias selbst erhält keinen Input. Er kann nützlich sein, wenn man eine Schwelle benötigt, die andere Inputs erst überschreiten müssen. Umgekehrt kann der Bias auch genutzt werden, um ein hoch aktives Neuron zu erzeugen, dass bereits bei geringen gewichteten Inputs aktiv wird. Die Funktionsweise des Perzeptrons wird in der nachfolgenden Abbildung 2-2 dargestellt.

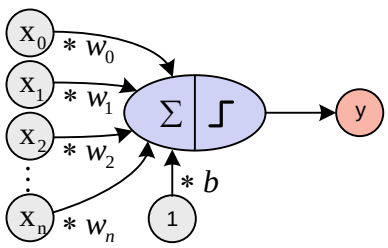



**Abbildung 2-2:** Single layer neuronal network

Für die Aktivierungsfunktion sind in der Praxis die folgenden Funktionen gebräuchlich: linear, binär, sigmoid, und Normalverteilung. Abbildung 2-3 stellt diese Aktivierungsfunktionen dar.

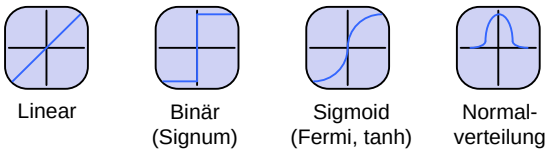



**Abbildung 2-3:** Typische Aktivierungsfunktionen

Die geeignetste Funktion ist vom jeweiligen Sachverhalt abhängig. Für einen linearen Zusammenhang beispielsweise ist normalerweise auch die entsprechende Aktivierungsfunktion zu wählen.

Wie bereits zuvor erwähnt, wiesen Minsky und Papert im Jahr 1969 nach, dass nur linear separierbare Funktionen mit einem single layer neuronal network (Perzeptron) erlernbar und auch die Anzahl der Schritte a priori nicht abschätzbar sind [MiPa69]. Demnach lässt sich die wichtige logische XOR-Funktion nicht mit einem Perzeptron erlernen.

Die folgende Abbildung 2-4 verdeutlicht die lineare Klassifikation (links) von Mustern [Bish06, 194ff.] und das XOR-Problem (rechts).

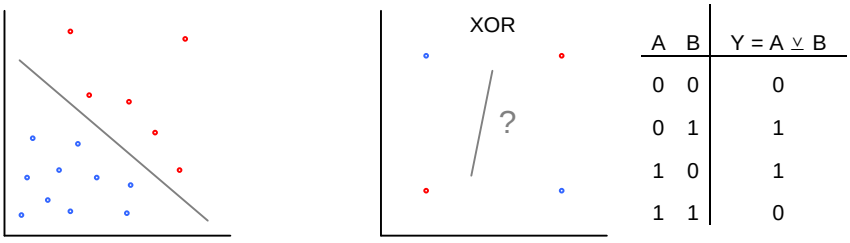



**Abbildung 2-4:** Lineare Separation von Mustern und XOR-Problem [Bish06, 195]

Durch diesen Sachverhalt wurden in den nachfolgenden 17 Jahren den Neuronalen Netzen weniger Aufmerksamkeit zugewandt, bis man im Jahr 1986 entdeckte, dass diese Aussage nicht für mehrschichtige Neuronale Netze (multi layer neuronal network) gilt. Der bereits erwähnte Backpropagation-Algorithmus von Rumelhart, Hinton und Williams kann alle denkbaren Funktionen lernen.

Das Verständnis eines einzelnen Knotens (Perzeptron) liefert die Grundlage für komplexere Neuronale Netzstrukturen, auf welche an dieser Stelle nicht weiter im Detail eingegangen werden soll. In Abbildung 2-5 sieht man ein mögliches mehrschichtiges Neuronales Netz mit den Eingabemustern $x_i$, den Eingangsneuronen $n_i$, einem von außen nicht sichtbaren verdeckten Neuron *h* (hidden) und der Ausgabe *y*. Jedes einzelne Neuron, auch das verdeckte, arbeitet nach dem Prinzip des Perzeptron. Hierbei kann jedes Neuron eine eigene Summier- und Aktivierungsfunktionen aufweisen.

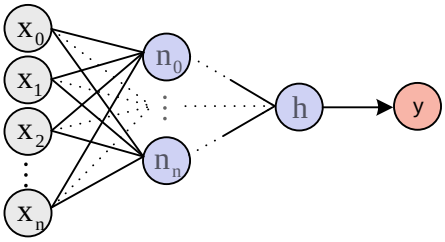



**Abbildung 2-5:** Multi layer neuronal network

Nach heutigem Stand der Wissenschaft lässt sich mittels Functional Combination auch mit einem Perzeptron das XOR-Problem lösen [YaBZ02]. Des Weiteren lässt sich durch *Mapping* in einen höherdimensionalen Raum und Separation durch eine Hyperfläche eine Vielzahl weiterer nicht linearer Probleme mit einem Perzeptron lösen. Dies sind jedoch rein mathematische Ansätze und haben mit dem biologischen Vorbild nur noch wenige Gemeinsamkeiten. Aber selbst die Funktionsweise eines Künstlichen Neuronalen Netzes ist von der eines Biologischen (noch) weit entfernt und eine andere Betrachtung als die Lösung eines medizinischen Problems in den Bereichen des Gehirns. Zwar können diese Modelle unterstützend wirken, aber die kognitiven Leistungen eines Menschen vermögen sie jedoch nicht zu erklären. Ferner wirkt die „Künstliche Intelligenz“ weiterhin nur auf gezielte Problemstellungen.

Nachfolgend wird nun auf die Komponenten und die Funktionsweise einer Grafikkarte eingegangen.

## 2.2 Funktionsweise einer Grafikkarte

Als im Jahr 1981 der erste *Personal Computer (PC)* der Firma *International Business Machines Corporation (IBM)* der Öffentlichkeit präsentiert wurde, existierte nur ein aus 80 Zeichen × 25 Zeilen bestehender Textmode. Trotz Bemühungen seitens des Grafikkartenherstellers Hercules, welcher für den *Monochrome Display Adapter (MDA)* einen Grafikmode entwickelte, wurde dieser Standard sehr schnell durch den *Color Graphics Adapter (CGA)* und anschließend im Jahre 1984 durch den *Enhanced Graphics Adapter (EGA)* abgelöst. Fünf Jahre später folgte der an Popularität und Bekanntheitsgrad unübertroffene *Video Graphics Array (VGA).*

Die damaligen Ausgabegeräte sind nur schwer mit heutigen Grafikkarten vergleichbar. Ihre Funktion beschränkte sich im Allgemeinen nur auf die Generierung des Ausgabesignals über die im Grafikspeicher – oder Hauptspeicher reserviertem Bereich – abgelegten Daten. Dazu wurde in der Regel ein Motorola 6845 [Moto77] oder baugleicher Grafiksteuerchip genutzt. Für die Umsetzung der ASCII-Zeichen im Textmode zu einer Punktmatrix ist der Zeichengenerator zuständig, welcher sich ebenso

auf der Grafikkarte befindet. Im Grafikmode ist dieser deaktiviert [Mess94, 16f.]. In nachfolgender Abbildung 2-6 wird die Funktionsweise einer EGA Grafikkarte aus den 1980er Jahren schematisch dargestellt.

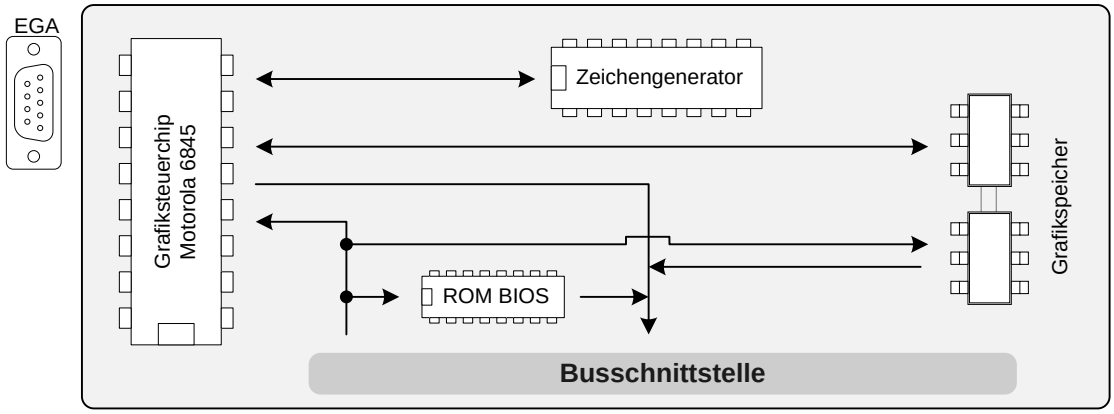



**Abbildung 2-6:** Blockdiagramm einer EGA Grafikkarte [Mess94, 16]

Um damaligen Anwendungen und vor allem den aufkommenden 3D-Spielen, wie z. B. Doom, gerecht zu werden, wurde der Ruf nach leistungsfähigerer Grafikhardware immer lauter. Dies geschah zunächst über sogenannte Beschleunigerkarten. Der Hersteller 3dfx war einer der ersten, welcher mit seinem *Voodoo Graphics Chipsatz* einen 3D-Beschleuniger mit eigenständiger GPU anbot. Durch eine eigenständige GPU wird die CPU massiv entlastet, indem z. B. das Zeichnen von Linien, das Füllen von Flächen und ähnlichem übernommen wird. Auch die Kopplung mehrerer Voodoo Karten zur Erhöhung der Rechenleistung war bereits möglich. Mit modernen Grafikkarten ist dieses auch heute noch möglich.

Im Vergleich zu einem Ausgabegerät, welches nur für die Generierung eines Ausgabesignals zuständig ist (vgl. Abbildung 2-6), funktionieren aktuelle Grafikkarten wie folgt: Daten und Befehle werden über den Bus an die GPU gesendet. Der Grafikprozessor errechnet ein digitales Bild, welches er im Grafikspeicher ablegt. Wird das Bild über VGA ausgegeben, so wandelt der *Random Access Memory - Digital/Analog Converter (RAM-DAC)* die digitalen Bilddaten in ein analoges Signal. Je nach Anzeigegerät wird dieses dann intern wieder in digital zurückgewandelt (Liquid Crystal Display, LCD) oder analog ausgegeben (Cathode Ray Tube, CRT-Monitor). Bei digitalen Anzeigegeräten empfiehlt sich daher der Anschluss z. B. via *Digital Visual Interface (DVI)* oder *High Definition Multimedia Interface (HDMI)*. In diesem Fall bleibt der RAM-DAC ungenutzt. Abbildung 2-7 veranschaulicht den Aufbau einer aktuellen Grafikkarte.

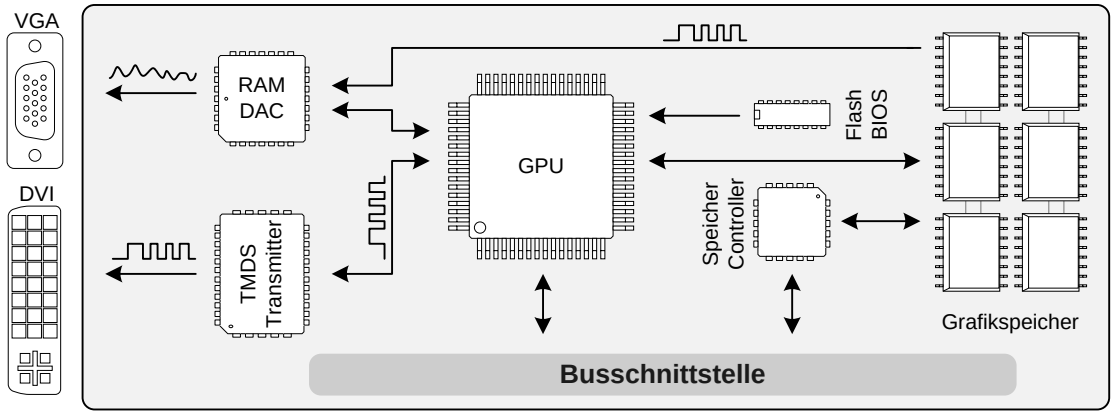



**Abbildung 2-7:** Blockdiagramm einer aktuellen Grafikkarte

Da viele Hersteller zu Einchiplösungen tendieren, sind nicht immer alle zuvor genannten Elemente physikalisch auf einer Grafikkarte vorhanden.

Noch einen Schritt weiter gehen *Integrated Graphics Processors (IGP)*. Während dedizierte, sprich nachträglich erweiterbare Grafikkarten, über Schnittstellen mit dem Bussystem verbunden werden, bezeichnet *onboard* fest verbaute und direkt mit dem Bussystem verdrahtete Grafikkomponenten. Aktuelle Grafikkarten nutzen den seriellen *PCI-Express (PCIe)*-Zugang. Timingprobleme bei hohen Geschwindigkeiten stoppten die Weiterentwicklung des parallel arbeitenden *Accelerated Graphics Port (AGP)*. AGP und PCIe sind Punktverbindungen zur Northbridge, während an der Southbridge weitere Controller angeschlossen sind. Die zuvor genannten IGP sind bereits so tief im System integriert, dass sie selbst ein Bestandteil der Northbridge sind. In Anlehnung an [Inte09b] zeigt Abbildung 2-8 die Systemarchitektur eines aktuellen PC, wie z. B. dem Intel Core 2 Duo.

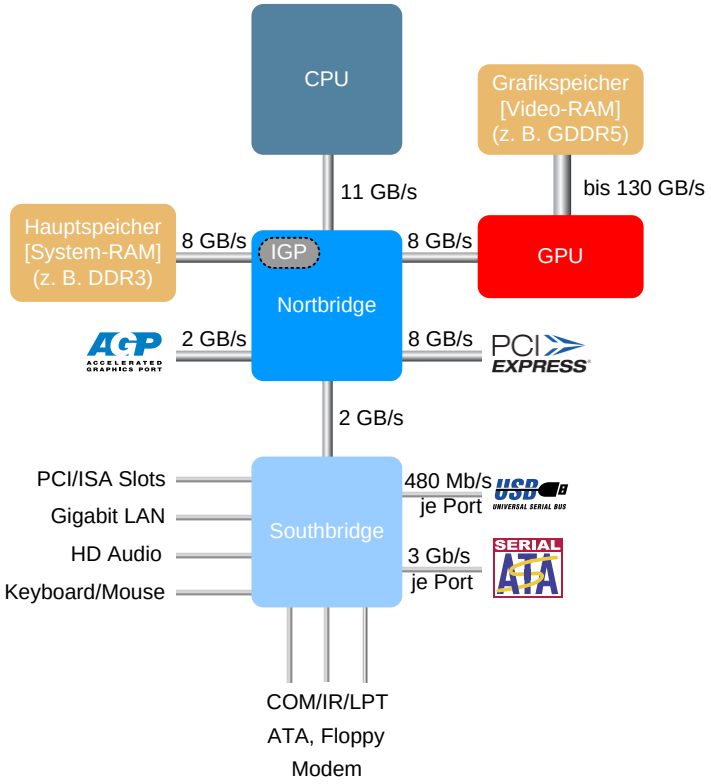



**Abbildung 2-8:** PC Systemarchitektur [Inte09b]

Interessant sind hierbei die derzeitigen Bandbreiten. Der Datentransport zur GPU stellt im Vergleich zum Grafikkarten-internen Datenaustausch einen Flaschenhals dar. Im Falle eines IGP und häufig auch bei onboard-Grafikkomponenten teilen sich aus Kostengründen CPU und Grafikprozessor den Hauptspeicher. Dieses Verfahren nennt man *Shared Memory*. Dem Grafikprozessor zugeordneter Speicher steht dem Betriebssystem nicht mehr zur Verfügung. Aufgrund der niedrigeren Bandbreite zum Hauptspeicher stehen Shared Memory-Systeme vollwertigen Grafikkarten bezüglich 3D-Rechenleistung um einiges nach. Bei 2D-Berechnungen (Desktopdarstellung etc.) sind jedoch keine Leistungsunterschiede bemerkbar.

Nachfolgend werden nun die Komponenten der Grafikkarte Grafikspeicher, RAM-DAC sowie DVI und TMDS näher betrachtet.

### 2.2.1 Grafikspeicher

Mit neuen Grafikkarten wachsen gleichzeitig die Ansprüche an den Grafikspeicher. Von *Fast Page Mode Dynamic RAM (FPM DRAM)* über *Extended Data Output RAM (EDO RAM)* zu *Video RAM (VRAM)*, welcher erstmals ein voneinander getrenntes I/O-Interface (Ein- und Ausgabekanal) besaß und somit gleichzeitiges Lesen und Schreiben zuließ.

Der derzeit aktuelle Grafikspeicher nennt sich *Graphics Double Data Rate, 5. Generation (GDDR5)* [Qimo07]. GDDR-RAM ist ein speziell für Grafikkarten entwickelter DDR-Speicher. Er besitzt zwei I/O-Interfaces, sodass CPU und GPU gleichzeitig lesen und schreiben können (vgl. Abbildung 2-7).

In erster Linie wird der Grafikspeicher für die Zwischenspeicherung des berechneten Bildes für die Ausgabe auf den Monitor benötigt. Die nachfolgende Gleichung (2.1) berechnet den theoretischen Bildspeicherbedarf (*Framebuffer*) für Darstellungen.

$$\begin{aligned} &\text{Bildspeicher} = \text{Auflösung} \times \text{Farbtiefe} \\ &\text{UXGA: } 1600 \times 1200 \times 32\ \text{bit} = 7{,}33\ \text{MB} \\ &\text{Anmerkung: } 32\ \text{bit} \Rightarrow 24\ \text{bit RGB } (2^{24}\ \text{Farben}) + 8\ \text{bit AlphaChannel} \end{aligned} \tag{2.1}$$

Betrachtet man jedoch die Größe des Grafikspeichers – von derzeit bis zu 2 GB – so ist ersichtlich, dass dieser seit Erfindung der Voodoo Karten nicht mehr nur als reiner Framebuffer dient. Die Größe des Grafikspeichers bestimmt somit nicht mehr die maximale Auflösung. 3D-Programme hinterlegen im Grafikspeicher z. B. ihre DirectX Hilfsroutinen, Geometrie- und Texturdaten. Ebenso belegt der *Z-Buffer* (Tiefenpuffer) einen nicht unbedeutenden Teil des Grafikspeichers. Während der Framebuffer den sichtbaren Teil des Bildes enthält, beinhaltet der Z-Buffer die Tiefe des sichtbaren Objekts an jedem Pixel. Dies wird zur Ermittlung verdeckter Flächen benötigt.

### 2.2.2 RAM-DAC

Wie zuvor kurz beschrieben, ist für die Generierung eines analogen Ausgabesignals der RAM-DAC verantwortlich. Vereinfacht betrachtet besteht dieser aus einem kleinem Festspeicher, in dem Farbtabellen abgelegt sind, dem Zeichengenerator für den Textmode (vgl. Abbildung 2-6) und drei Highspeed D/A-Wandlern (Digital/Analog), da jeder Pixel aus drei Subpixeln der Grundfarben *Rot, Grün, Blau (RGB)* dargestellt wird. Die Farbtabellen sind aus Gründen der Abwärtskompatibilität noch vorhanden, werden heutzutage aber nicht mehr benötigt. Der Programmierer konnte im 256-Farben-Zeitalter mittels dieser Tabellen die Farben anpassen.

Die Arbeitsweise des RAM-DAC kann wie folgt beschrieben werden: An die GPU übermittelt der RAM-DAC den Takt und bekommt von der GPU Blank- und Sync-Signale für den Monitor. Zusätzlich hat er direkten Zugriff auf jenen Teil des Grafikspeichers, der als Framebuffer dient. Pixel für Pixel wird so der Farbwert aus dem Speicher geholt und mittels Multiplexer in R, G, B getrennt und in den jeweiligen D/A-Wandler „geschoben“. Dieser rechnet den Wert in eine analoge Spannung um und legt diese auf die jeweilige R-, G- oder B-Leitung auf.

Das nachfolgende Blockdiagramm [Anal95, 1] (Abbildung 2-9) veranschaulicht die eben genannten Komponenten, sowie die Funktionsweise des RAM-DAC.

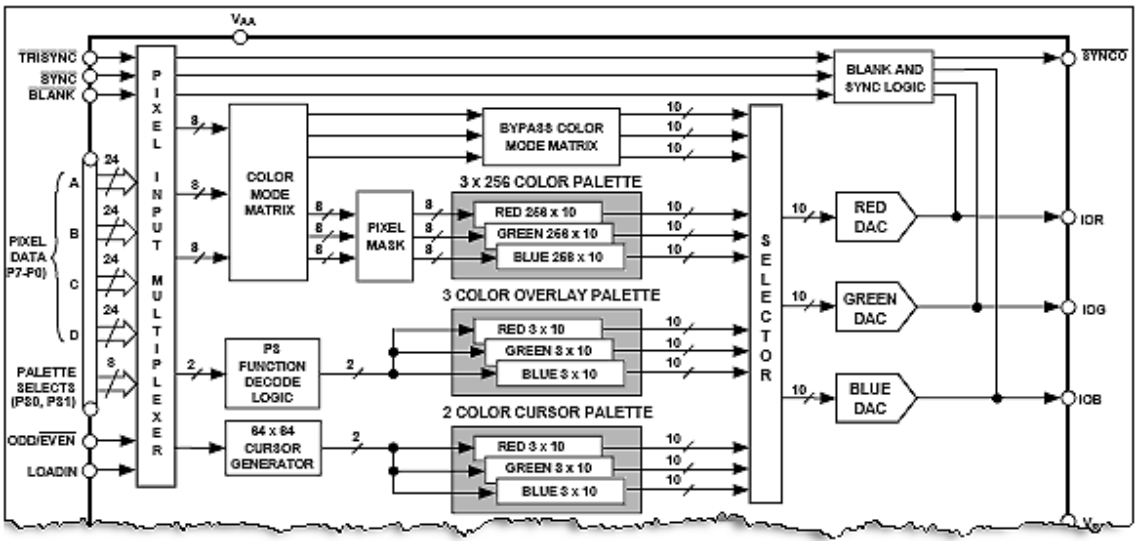



**Abbildung 2-9:** Blockdiagramm eines RAM-DAC [Anal95, 1]

In den Konfigurationseinstellungen einiger Grafikkarten ist die Option *Downfilter at Scanout* vorhanden. Bei dieser werden die RGB-Subpixel und Anti-Aliasing (Kantenglättung) im Framebuffer hinterlegt. Diese Einstellung erspart der GPU das Mischen und Schreiben des finalen Pixels, vergrößert aber den Framebuffer. Bei vierfachem Anti-Aliasing wird dieser demzufolge um das Vierfache vergrößert! Die folgende Abbildung 2-10 zeigt einige mögliche Grafikoptionen.

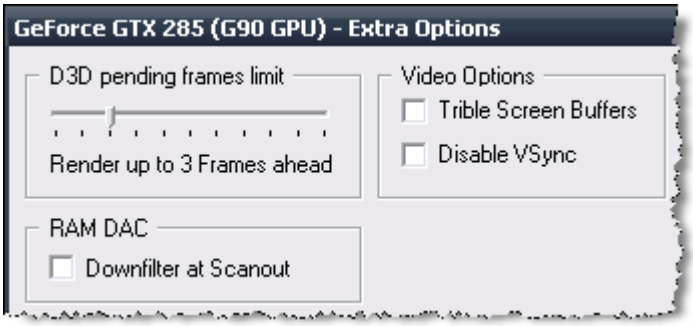



**Abbildung 2-10:** Grafikoptionen (Symbolfoto)

Beim genaueren Betrachten der Arbeitsweise des RAM-DAC fällt auf, dass die GPU in den Framebuffer schreiben kann, während der RAM-DAC noch bei der Umwandlung ist. Dies zieht bei schnellen Szenen Flimmern und Artefaktebildung nach sich, da eine Grafik normalerweise durch Löschen und Neuzeichnen entsteht. Deshalb wird intern meistens mit *double buffering* (zweifacher Framebuffer) gearbeitet. Es gibt zwei Puffer: *Frontbuffer* und *Backbuffer*. Während der RAM-DAC mit dem Frontbuffer arbeitet, wird der Backbuffer gefüllt. Danach wird ein sogenannter *swap* durchgeführt, bei dem

die Zugriffsadressen vertauscht werden. Das bedeutet, dass der Frontbuffer jetzt der Backbuffer ist und umgekehrt. Der RAM-DAC arbeitet jedoch einfach weiter, da er von dem Wechsel nichts mitbekommen hat. Bei schnellen, bewegten Szenen kann dies zu *tearing* (Verschiebung) innerhalb des Bildes führen.

Teils wird dies als störend empfunden. Hier loht sich ein Blick auf die Option *VSync*. In diesem Fall wartet die GPU mit dem swap bis der RAM-DAC signalisiert, dass er mit der Wandlung der letzten Bildzeile fertig ist.

„Jede Lösung eines Problems ist ein neues Problem“ sagte einst Johann Wolfgang von Goethe. Leider ist dies bei VSync nicht anders. Durch gegenseitige Wartezeiten ist die Framerate (Bildrate) immer ein Vielfaches der Bildwiederholfrequenz. Bei 100 Hz Bildwiederholfrequenz können maximal 100 Frames pro Sekunde (fps) ausgegeben werden, auch wenn Grafikhardware deutlich mehr schaffen könnte. Das größere Problem ist aber, wenn die GPU nur 99 fps schafft. In diesem Fall sinkt die tatsächliche Rate auf 100/2 = 50 fps, da der RAM-DAC immer zweimal aufeinanderfolgend dasselbe Bild ausgeben muss, weil die GPU nicht fertig war. Diese kann wiederum nicht weiterarbeiten, weil auf swap gewartet werden muss. Sinkt die theoretische Rate auf unter 50 fps, sind es effektiv nur noch 100/3 = 33 fps usw. Die nachfolgende Rechnung (2.2) verdeutlicht die VSync-Problematik.

$$
\begin{aligned}
100\ \text{Hz} &= \frac{1\text{s}}{100} \\
&\Rightarrow \text{alle 10 ms Bildwiederholung} \\
99\ \text{fps} &= \frac{1\text{s}}{99} = 10{,}1\ \text{ms Berechnungsdauer} > 10\ \text{ms} \\
&\Rightarrow \text{RAM DAC gibt 2× selbes Frame aus} \\
49\ \text{fps} &= \frac{1\text{s}}{49} = 20{,}4\ \text{ms Berechnungsdauer} > 2 \cdot 10\ \text{ms} \\
&\Rightarrow \text{RAM DAC gibt 3× selbes Frame aus}
\end{aligned}
\tag{2.2}
$$

Der massive Performanceverlust bei Unterschreiten einer Grenze ist ein großes Problem bei dieser Einstellung. Um diesem Effekt entgegen zu wirken, gibt es oftmals die Option *triple buffering* (dreifacher Framebuffer). Der größere Framebuffer-Speicherverbrauch lässt sich angesichts heutiger Speichergrößen verkraften. Auch hier ist die maximal mögliche Rate identisch mit der Bildwiederholfrequenz. Die *Pre-Render*-Option könnte in diesem Kontext zu etwas Verwirrung führen, da sie lediglich angibt für wie viele Frames im Voraus die GPU Befehle annimmt. Dies entkoppelt CPU und GPU, kann bei Spielen aber zu sogenannten *mouse lags*, also der Asynchronität zwischen Bild und Eingabe, führen. Auf andere Optionen soll an dieser Stelle nicht weiter eingegangen werden.

Typischerweise arbeiten RAM-DAC aktuell mit 400 MHz. Was diese Zahl für die Praxis bedeutet, veranschaulicht folgendes Beispiel (2.3).

$$\begin{aligned} &\text{Pixelfrequenz} = \text{Auflösung} \times \text{vert. Wiederholfrequenz} \times \text{Overhead} \\ &\text{(CRT) Overhead} \approx \sqrt{2} = 1{,}41 \\ &\text{UXGA: } 1600 \times 1200 \times 100\ \text{Hz} \times \sqrt{2} \approx 269\ \text{MHz} \end{aligned} \tag{2.3}$$

Der in der Gleichung verwendete Overhead in Höhe von 1,41 ist ein allgemeiner Rundungswert. Er ergibt sich aus dem nicht sichtbaren Randbereich (Overscan) bei Röhrenmonitoren und der Zeit für Abschalt- und Rücklaufdauer des Elektronenstrahls nach Zeilenende (Zeilenaustastlücke) und Bildende. Bei einer Auflösung von $1600 \times 1200 \times 100$ Hz ergibt sich lediglich eine Leistungsreserve von 131 MHz für RAM-DAC, die mit 400 MHz arbeiten. Aus diesem Grund könnte ein zweiter Monitor nicht gleichzeitig mit selbiger Auflösung und vertikaler Bildwiederholfrequenz betrieben werden. Um dieses Problem zu lösen, werden oftmals zwei RAM-DAC verbaut. Hier lohnt im Einzelfall genaues Hinsehen!

### 2.2.3 DVI und TMDS

Eine zum VGA-Anschluss zeitgemäße Alternative stellt der, von der Digital Display Working Group[1] ins Leben gerufene, DVI-Anschluss dar [Ddwg99]. Dieser ist praktisch auf jeder aktuell erhältlichen Grafikkarte vorhanden. Für die digitale Ausgabe wird der RAM-DAC nicht benötigt, wodurch das Bild verlustfrei übertragen werden kann. Wirklich neu ist dieser Trend jedoch nicht, denn bis zur Einführung des VGA war die Übertragung digital (9 pol. SUB-D Buchse, vgl. Abbildung 2-6). Damals war es allerdings nicht möglich mit wenigen Leitungen feine Farbabstufungen digital zu übertragen. Durch schnelle Chips und neue Codierungsverfahren wird dies jetzt jedoch ermöglicht.

In Notebooks ist aus Platz- und multimedialen Gründen statt DVI häufig ein HDMI-Anschluss [Hdmi06], welcher aus der Unterhaltungsindustrie stammt, vorzufinden. Da diese Industriesparte sehr empfindlich auf das Thema „Privatkopie“ reagiert, sei der Vollständigkeit halber erwähnt, dass bei fast allen Entertainment-Geräten das Kopierschutzverfahren *High-bandwidth Digital Content Protection (HDCP)* [Hdcp06] zum Einsatz kommt. HDMI kann neben digitalen Bildinhalten gleichzeitig digitalen Ton übertragen.

[1] Zusammenschluss von Fujitsu, Compaq, HP, IBM, Intel, NEC und Silicon Image. http://www.ddwg.org.

Sowohl DVI als auch HDMI verwenden das *Transition Minimized Differential Signaling-Protokoll (TMDS-Protokoll)* und sind daher zueinander kompatibel. Der derzeitige DVI-Standard sieht für den TMDS-Transmitter (Sender) eine Pixelfrequenz von minimal 25,175 MHz und maximal 165 MHz vor. Mittels nachfolgender Gleichung (2.4) kann die maximale und minimale Prixelfrequenz berechnet werden.

$$\text{Pixelfrequenz} = \text{Auflösung} \times \text{vert. Wiederholfrequenz} \times \text{Overhead} \qquad (2.4)$$

$$\text{Overhead} = 1 + \frac{\left(\frac{\text{Blanking}}{1 - \text{Blanking}}\right) \text{ in \%}}{100}$$

Aus dieser Gleichung ergibt sich im Single-Link eine minimale Auflösung von 640×480 und eine maximale Auflösung von 1600×1200 (UXGA) bzw. 1920×1200 Pixel (WUXGA) bei jeweils 60 Hz, wenn Monitor und Grafikkarte ein sogenanntes *reduced blanking*, das die Verkürzung der Abschalt- und Rücklaufzeiten für LCD ermöglicht (vgl. Abschnitt 2.2.2), unterstützen [Sili04, 20].

Mittels Dual-Link liegt die derzeit von DVI unterstützte Auflösung bei 2560×1600 mit 60 Hz. Dafür werden zwei TMDS-Transmitter auf der Grafikkarte benötigt! Dies impliziert, dass nicht jede Grafikkarte mit zwei DVI-Ports automatisch Dual-Link fähig ist.
Für noch höhere Auflösungen werden vier TMDS-Transmitter bzw. zwei Dual-Link-Anschlüsse benötigt, was allerdings nur von sehr wenigen Spezial-Grafikkarten und Bildschirmen unterstützt wird. Zu nennen wäre hier die FireGL Serie von ATi oder die Quadro FX Serie von nVIDIA. Die folgende Abbildung 2-11 stellt die Übertragung von Halbbildern bei hohen Auflösungen grafisch dar.

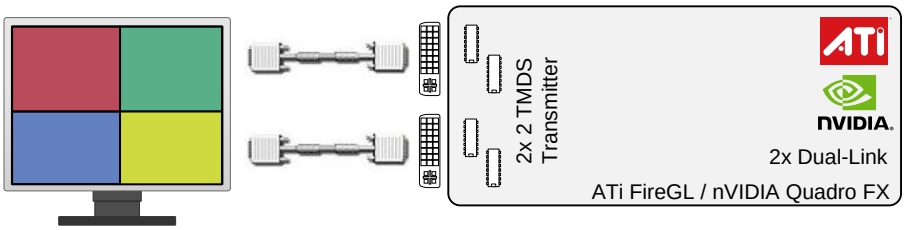



**Abbildung 2-11:** DVI zweifach Dual-Link

Mit *selective refresh* soll in zukünftigen Versionen die benötigte Bandbreite reduziert werden, indem nur der geänderte Inhalt zum Monitor übertragen wird. Dies setzt einen von außen adressierbaren Speicherbereich im Monitor voraus [Ddwg99, 8].

Für einen kurzen Einblick in die Funktionsweise wird nun der DVI Link betrachtet. Der DVI Single-Link besteht aus drei Datenkanälen (RGB) und einem Kanal, der den Takt

überträgt. Bei Dual-Link sind es dementsprechend sechs Datenkanäle. In Abbildung 2-12 wird dies dargestellt.

Ein Codieralgorithmus wandelt die 8 bit parallel anliegenden Daten auf jedem Kanal in 10 bit serielle und überträgt diese zum Monitor. Intern arbeitet der TMDS-Chip demzufolge mit zehnfacher Geschwindigkeit. Die 8b/10b-Codierung dient der Übertragungssicherheit gegenüber Störquellen. Im TMDS-Receiver findet folglich der umgekehrte Prozess der Codierung statt. Auf eine detaillierte Darstellung des Codieralgorithmus wird an dieser Stelle verzichtet. Eine ausführlichere Darstellung des Codieralgorithmus findet man in den Datenblättern von [Ddwg99, 28ff.; Sili04, 12ff.].

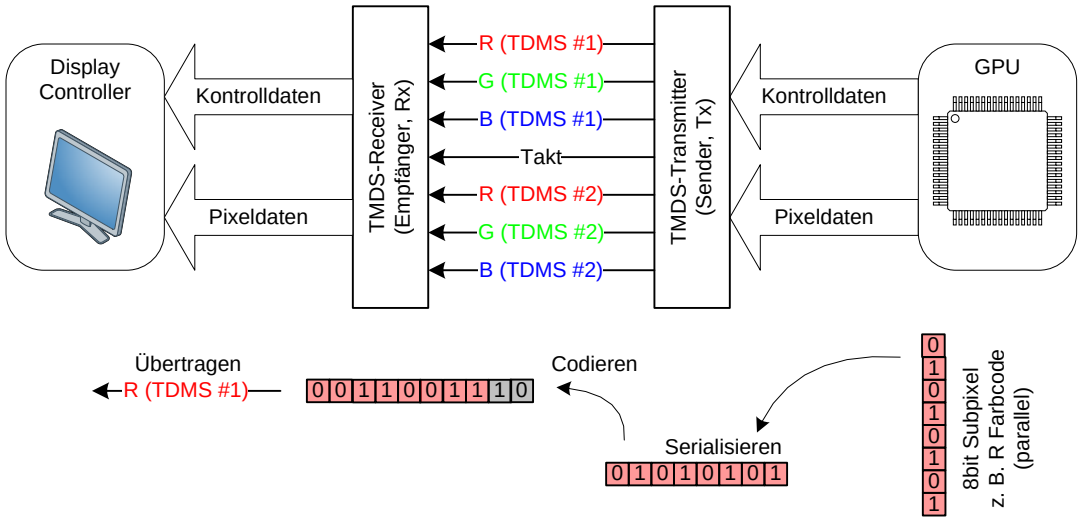



**Abbildung 2-12:** TMDS Logical Links und Codierung [Ddwg99, 10]

Neben DVI und HDMI existieren noch zwei weitere, mehr oder weniger verbreitete, Multimediaschnittstellen: *DisplayPort* und *Unified Display Interface (UDI)*, wobei letzteres als gescheitert gilt. *High Definition Serial Digital Interface (HDSDI)* ist eine Eindrahtlösung (Koaxkabel) und findet nur im professionellen Video-Umfeld Anwendung. Hierbei werden die RGB- und Audiosignale ineinander verschachtelt und in einem einzelnen seriellen Stream übertragen.

Für die Interkonnektivität existieren entsprechende aktive (mit interner Elektronik) oder passive Adapter. Ein passiver DVI- zu VGA-Adapter funktioniert natürlich nur, wenn die RAM-DAC VGA-Ausgänge auf bestimmte Pins im DVI-Anschluss aufgelegt sind. Im DVI-Standard sind für die analoge Nutzung extra Pins definiert.

Bei den digitalen Anschlüssen besteht die gleiche VSync-Problematik wie bei den zuvor beschriebenen analogen Schnittstellen (siehe Abschnitt 2.2.2).

Abschließend gilt noch zu bemerken, dass ein LCD ab Werk eine optimale und gleichzeitig maximale Auflösung vorgegeben hat. Diese nennt sich *Native Auflösung*.

Eine niedrige Auflösung kann ein LCD durch Streckung des Bildes, was oftmals verschwommen wirkt, oder durch Wiedergabe in der exakten Auflösung mit schwarzen Rändern darstellen. LCD gelten bereits ab 60 Hz flimmerfrei, da die Pixel-Nachleuchtdauer elektronisch geregelt wird und nicht auf Phosphoreszenz beruht. Weiterhin sind sie strahlungsärmer als CRT-Monitore, aber nicht strahlungsfrei! Die Möglichkeit der Rekonstruktion des Bildes über einige Meter Luftlinie besteht bei beiden Systemen [Kuhn03].

Nachfolgend soll nun auf die GPU, als Komponente der Grafikkarte, näher eingegangen werden.

## 2.3 Status Quo der GPU

Der kommerzielle Grafikchip-Markt wird durch die Spiele-, Automobil- und Entertainment-Industrie angetrieben. Doch vor allem durch die große Nachfrage auf dem Spielemarkt erlebte die GPU einen rasanten Performanceschub mit zeitgleich günstigen Preisen. Im Folgenden wird daher die Marktentwicklung, der Stand der Technik im Bereich der Grafikchipsätze sowie die Performanceentwicklung von GPU untersucht.

### 2.3.1 Marktverteilung

Der Grafikkartenmarkt gehört zu einem der weltweit wettbewerbsintensivsten Märkte. Im ersten Quartal 2009 wurden insgesamt 74,87 Millionen Grafikeinheiten verkauft, 21,1 % weniger als im vierten Quartal 2008. Dies ist unter anderem auf die weltweite Wirtschaftskrise zurückzuführen. Das zum Jahresende 2009 erschienene Microsoft Windows 7 und DirectX 11 könnte das Geschäft jedoch wieder ankurbeln [Pedd09]. Neben dem klassischen Geschäftsfeld steht derzeit das *Stream Computing* oder auch *GPGPU* (siehe Abschnitt 2.4), also die Nutzung der GPU für mathematische Zwecke, im Rampenlicht. Versprochen wird viel Rechenleistung zu kleinen Preisen, was neue Käuferkreise aus Wissenschaft und Technik akquirieren soll.

Wer auf dem Markt für Grafikkarten überleben möchte, muss mit der Schnelllebigkeit des Marktes mithalten oder alternative Konzepte anbieten. Aus diesem Grund ist es nicht wunderlich, dass sich der Markt für Grafikkarten konsolidiert und Firmen entweder aufgekauft werden oder vom Markt austreten. Zum Teil erfolgt dies durch komplette Geschäftsaufgabe oder durch Konzentration auf andere Geschäftsfelder. So wurde z. B. Hercules im Jahr 1999 von Guillemot aufgekauft und produziert aktuell Webcams, Lautsprecher und Wireless Produkte. Im Jahr 2000 kaufte nVIDIA die Firma 3dfx auf und erwarb alle Patente. Im Jahr 2006 übernahm der Prozessorhersteller

Advanced Micro Devices (AMD) den GPU-Hersteller ATi Technologies für rund 4,3 Mrd. Euro [Kauf06]. ATi wird seither als Markenname für diverse Grafikkarten-produkte von AMD weiterverwendet.

Allgemein wird der PC GPU-Markt durch die drei Firmen Intel, nVIDIA und AMD (ATi) beherrscht. Angeführt wird der Markt dabei durch den weltweit größten Chiphersteller Intel mit einem Marktanteil von fast 50 %. Intel produziert IPG für Notebook, Server und Desktopeinsatz und bietet keine Chipsätze für dedizierte Grafikkarten an. Auf Platz zwei steht aktuell die Firma nVIDIA, welche ungefähr ein Drittel des Marktes bedient. ATi besitzt als drittgrößter GPU-Hersteller einen Marktanteil von rund 17 %. Sowohl ATi als auch nVIDIA bieten High-End Grafikchipsätze für Spieler und professionelle Designer im PC-Umfeld. Ebenso produzieren sie die Grafikchipsätze für Notebooks und Spielkonsolen wie Sony PlayStation 3 ⟨nVIDIA⟩, Microsoft XBox 360 ⟨ATi⟩, Nintendo Wii ⟨ATi⟩. Die Architektur einer solchen Konsole unterscheidet sich kaum noch von der eines PC (siehe Abbildung 2-8).

Die drei weiteren Firmen VIA/S3, SiS und Matrox bedienen zusammen den verbleibenden Marktanteil von insgesamt 2,1 %. Das Joint-Venture VIA Technologies / S3 Graphics produziert hierbei für den Embedded-Markt und versucht sich seit dem Jahr 2003 wieder im Desktop-Bereich zu etablieren. Im Jahr 2006 lag der Marktanteil noch bei 6,7 % [Pedd07]. Dieser ist jedoch im ersten Quartal des Jahres 2009 weiter auf 1,10 % gesunken. Die Firma Silicon Integrated Systems (SiS) stellte noch bis zum Jahr 2006 Grafikchips für den Desktop- und Notebook-Markt her. Aktuell zieht sich SiS jedoch aus dem GPU-Markt zurück und nur noch die SiS-Tochter XGI Technology Inc. konzentriert sich auf den Server- und Embedded-Markt.

Den mit Abstand geringsten Marktanteil hat die Firma Matrox Graphics. Diese bedient einen Nischenmarkt und bietet gegenwärtig Grafiklösungen für den professionellen Einsatz an. Die Produkte werden hauptsächlich noch dort eingesetzt, wo längerfristig mit einem Hardwarebestand gearbeitet werden muss, wie z. B. Börsen- und Kassenterminals und medizinische Displays. Durch hohe Anschaffungskosten müssen hier längere Nachkaufzeiten garantiert werden.

Abbildung 2-13 verdeutlicht die Marktanteile der zuvor genannten GPU-Hersteller grafisch.

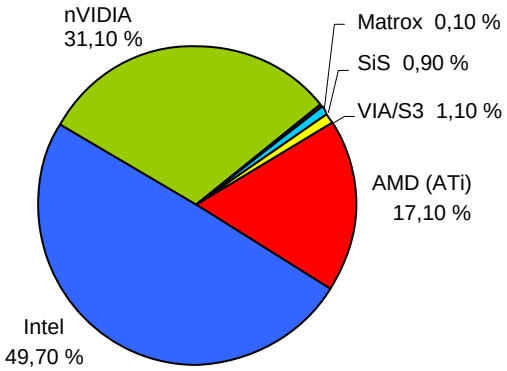



**Abbildung 2-13:** PC GPU-Marktanteile Q1/2009

Im Folgenden wird auf die Entwicklung und den aktuellen Stand der Technik von modernen Grafikkarten eingegangen.

### 2.3.2 Stand der Technik

Mittlerweile existieren GPU seit Mitte der 1990er Jahre. Bis heute haben sich diese stetig weiterentwickelt und lassen sich mittlerweile in sechs Generationen einteilen. Tabelle 2-1 zeigt hierzu eine Übersicht dieser sechs GPU-Generationen.

**Tabelle 2-1:** GPU-Generationen

| |
|---|
| *Pre-GPU (bis 1995)*<br>Die CPU berechnete Bild, die Grafikhardware war für Generierung des Ausgabesignals verantwortlich. Professionelle Anwender mussten auf teure Spezialhardware z. B. von Silicon Graphics (SGI) zurückgreifen. Die heutige GPU-Generation ist viel günstiger und leistungsfähiger als damalige Spezialhardware. |
| *1. GPU-Generation (ab 1995)*<br>DirectX fähige Hardware wie: nVIDIA TNT2, ATi Rage, 3dfx Voodoo. Limitierter mathematischer Befehlssatz, einfaches Texturmapping und Z-Buffer. Transformation (3D-Objekte verschieben und drehen) weiterhin via CPU. |
| *2. GPU-Generation (ab 1999)*<br>DirectX 7 fähige Hardware wie: nVIDIA GeForce 256 und GeForce 2, ATi Radeon 7500, S3 Savage 3D. GPU Multitexturmapping, Transformation und lighting (Lichteinstrahlung) möglich. GPU weiterhin nicht direkt programmierbar. |
| *3. GPU-Generation (ab 2001)*<br>DirectX 8 fähige Hardware wie: nVIDIA GeForce 3 und GeForce 4, Microsoft Xbox, ATi Radeon 8500. Programmierbarer Vertex-Shader (fügt Objekten Spezialeffekte zu). Erweitert, jedoch nicht vollständig, Pixelprogrammierbar. |
| *4. GPU-Generation (ab 2002)*<br>DirectX 9 fähige Hardware wie: nVIDIA GeForce FX bis GeForce 7, ATi Radeon 9700. Vertex- und Pixel-Shader voll programmierbar. GPGPU prinzipiell möglich (siehe auch Abschnitt 2.4). |
| *5. GPU-Generation (seit 2006)*<br>DirectX 10 fähige Hardware wie nVIDIA GeForce 8 und GTX 285, Microsoft Xbox 360, Sony Playstation 3, ATi Radeon HD. Shader model 4 (Unified Shader) und 64 bit Farbtiefe. |
| *6. GPU-Generation (Einführung Q4/2009)*<br>DirectX 11 fähige Hardware. Shader model 5 (Compute Shader). |

Um den gesamten Markt von Einsteigern, Fortgeschrittenen, Spielern bis hin zu professionellen Anwendern bedienen zu können, wird ein breites Spektrum an verschiedenen GPU-Modellen angeboten. Ergänzend bieten die Hersteller ATi und nVIDIA auch spezielle Grafikkarten-Versionen für *High Performance Computing (HPC)* an. Bei ATi ist dies die FireStream und bei nVIDIA die Tesla Serie. Die zuletzt genannten Modellreihen sind Grafikkarten ohne Monitorausgang und überzeugen mit deutlich mehr Arbeitsspeicher. Des Weiteren hat nVIDIA bereits mit Tesla Hardware bestückte Workstations und Rack-Einschübe im Programm.

In folgender Tabelle 2-2 werden die Oberklassen der derzeit erhältlichen GPU, bzw. der darauf basierenden Grafikkarten, beider Firmen gegenübergestellt[2].

**Tabelle 2-2:** Technische Daten der aktuellen ATi und nVIDIA High-End GPU

| | ATi<br>Radeon HD 4890<br>FireStream 9270 | nVIDIA<br>GeForce GTX 285<br>Tesla C1060 |
|---|---|---|
| GPU-Modell Nr. | RV790<br>(RV770 FireStream 9270) | GT200b |
| Jahr der Markteinführung | 2009 (2008 FireStream 9270) | 2009 |
| GPU-Takt | 850 MHz | 648 MHz |
| Stream Prozessoren (SP) | 160 5D Vektor SP<br>(siehe Abschnitt 2.4.2) | 240 skalare SP |
| Theoretische Leistung | 1360 Gflops (single precision) | 1063 Gflops (single precision) |
| Grafikspeicher | 1 GB (GDDR5)<br>(2 GB GDDR5 FireStream 9270) | 1 GB (GDDR3)<br>(4 GB GDDR3 Tesla C1060) |
| RAM-Takt | 1950 MHz | 1242 MHz |
| Speicheranbindung | 256 bit | 512 bit |
| Bandbreite | 124,8 GB/s | 158,9 GB/s |
| Leistungsaufnahme | ∅ 190 VA | ∅ 185 VA |
| Preis (ca.) | 210 €<br>(ca. 1.200 € FireStream 9270) | 320 €<br>(1.500 € Tesla) |

Über ein Brückenkabel können bis zu vier Grafikkarten eines Herstellers miteinander verbunden werden. Bei ATi nennt sich diese Technik *Crossfire*, bei nVIDIA *Scalable Link Interface (SLI)*.

Ähnlich der Tesla oder FireStream Serie bietet IBM eine PCIe Beschleunigerkarte auf Cell-Basis für PC und Rack an. Die rund 5.000 Euro teure PowerXCell-8i Erweiterungskarte hat einen Standardkern als Manager und acht auf parallele Verarbeitung optimierte Recheneinheiten, welche mit 2,8 GHz getaktet sind. Mit 4 GB DDR2 ECC RAM ausgerüstet schafft das Speicherinterface einen Durchsatz von 25 GB/s. Die theoretische Performance liegt bei 180 Millionen Floating-Point Operations per Second (Gflops) (single precision, einfache Genauigkeit). Schnittstellen nach außen sind zwei Gigabit Ethernet-Ports. Neben schneller paralleler Verarbeitung von `float`-Daten eignet sich diese Karte auch für Verarbeitung von Pixel-Daten, z. B. im Zusammenspiel der mvHYPERION-CLf Frame-Grabber-Karte.

Hohen Bekanntheitsgrad erlangte der Cell-Prozessor durch die Sony Playstation 3, in der dieser als CPU eingesetzt ist. Somit eignet sich die für den Massenmarkt entwickelte

[2] Stand vom 01.09.2009

Spielkonsole ebenso für HPC Test- und Entwicklungsumgebungen [Epfl09]. Ein entsprechendes Hintergrundwissen wird hierbei vorausgesetzt.

In typischen HPC-Rechenzentren werden derzeit Cluster aus Mehrkern-CPU-Systemen, wie z. B. dem Intel Xeon oder IBM Cell, gebildet. Industrie und Forschung werden jedoch sehr über die aktuellen GPU-Entwicklungen erfreut sein, da ein Standard-PC mit einer Radeon oder GeForce Karte für rund 1.000 Euro eine teure Dual-CPU-Workstation ersetzen könnte. Nachfolgend wird daher die Performanceentwicklung von Grafikkarten näher betrachtet.

### 2.3.3 Performanceentwicklung

Intel Mitgründer Gordon Earle Moore beobachtete bereits im Jahr 1965, dass sich die Komplexität integrierter Schaltkreise mit minimalen Komponentenkosten etwa jährlich verdoppelt [Moor65]. In der korrigierten Version des Mooreschen „Gesetzes“ aus dem Jahre 1975 [Moor75] ist von etwa allen zwei Jahren die Rede. Diese korrigierte Version des Mooreschen Gesetzes besitzt noch heute seine Gültigkeit.

Das Mooresche Gesetz ist keine direkte Vorschrift, sondern in erster Linie ein wirtschaftlicher Aspekt. Somit ist es nicht verwunderlich, dass es auch bei GPU Anwendung findet, wenn auch durch den hohen Konkurrenzdruck jährlich Innovationen geliefert werden müssen. Die nachfolgende Abbildung 2-14 veranschaulicht das Mooresche Gesetz bezogen auf die Entwicklung der GPU in dem Zeitraum von 1998 bis 2008 (Daten siehe Anhang E). Neben der Leistungserhöhung werden in Zukunft energieeffiziente Produkte eine weitere bedeutende Rolle spielen.

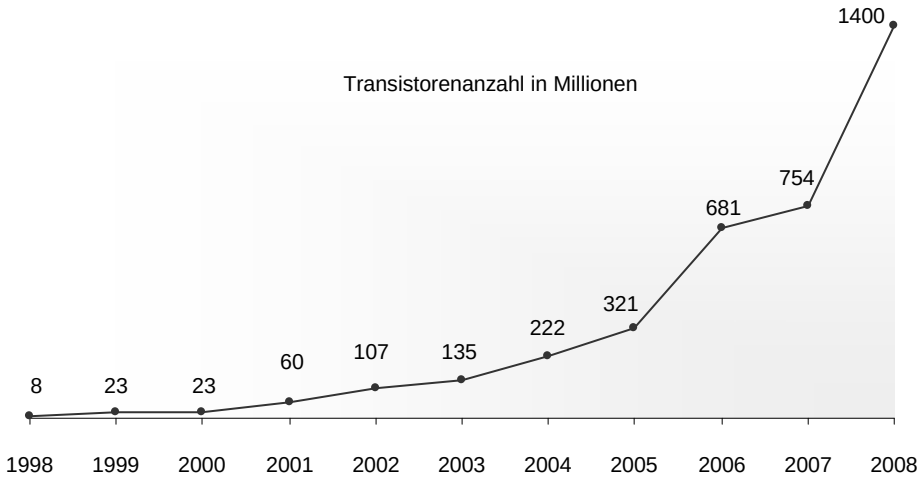



**Abbildung 2-14:** GPU-Zeitlinie Mooresches „Gesetz“

Die dargestellte Erhöhung der Komplexität ist in einem Computer gleichzusetzen mit der Erhöhung der Transistorenanzahl, da Transistoren derzeit das vorherrschende Bauelement sind. Eine Verdopplung der Transistorenanzahl ist allerdings nur vorteilhaft, wenn zugleich die Transistorendichte erhöht wird. Als Index dient hierzu der Fertigungsprozess, also die Größe eines Transistors. Auch hier unterliegen GPU, wie auch CPU, den allgemeinen Fertigungstechnologien. Die nachfolgende Abbildung 2-15 zeigt hierzu den Fertigungsprozess von GPU in dem Zeitraum von 1998 bis 2009 (Daten siehe Anhang E).

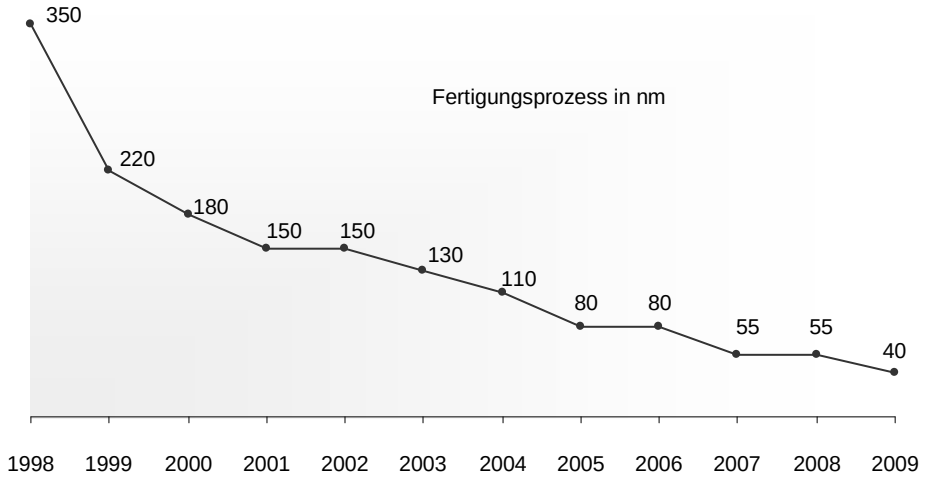



**Abbildung 2-15:** GPU-Zeitlinie Prozess Technologie

Während der erste Transistor-Prototyp von Bell Laboratories in den 1950er Jahren ein handgroßes Gebilde war, so werden demnächst GPU im 40 nm (0,00004 mm) Prozess gefertigt. Dies bedeutet, dass der Abstand zwischen Source und Drain eines einzelnen Transistors 40 nm ist.

Die meisten Größen im Micro- und Makrokosmos sind für uns Menschen jedoch sehr schwer vorstellbar. Bildlich verdeutlicht, würden mehr als 33.000 dieser Transistoren auf einen Stecknadelkopf passen, mehr als 2.200 Transistoren entsprechen dem Durchmesser eines Menschenhaares. Ein paar hundert solcher Transistoren könnte man in eine menschliche Zelle packen, wenn man denn möchte. Bei einem Fertigungsprozess von 40 nm ist ein einzelner Transistor in etwa so groß wie ein Hepatitis B Virus. Abbildung 2-16 soll hierbei helfen die Größenverhältnisse zu veranschaulichen.

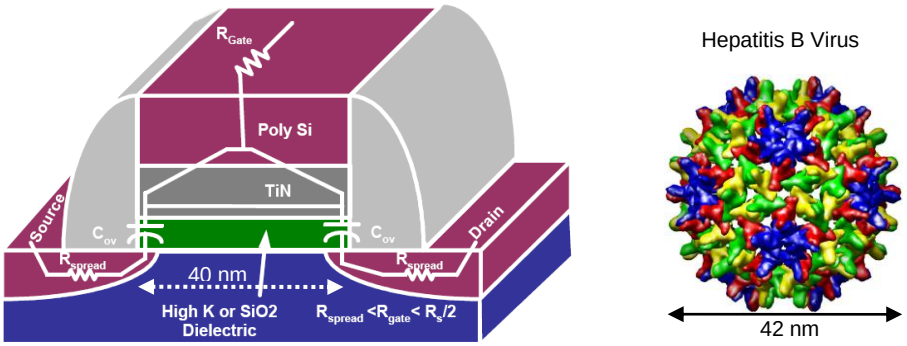



**Abbildung 2-16:** Größenvergleich Transistor [BAD+01] und Virus [Ucsf04]

Die durch Miniaturisierung entstehenden Leckstöme und bleifreies Löten sowie Wärmeentwicklung fordern Chip- und Chip-Gehäusehersteller gleichermaßen [PWA+08]. So wurde bekannt, dass einige Chip-Gehäuse von nVIDIA unter thermischer Belastung zu Ausfällen neigen [WiBL08]. Betroffen sind Chipsätze des Types G86, G86A2, G84, C51, G72, G72M, G73, G72A3, MCP67 und NV42 (vgl. Typen Anhang E).

Mit zunehmender Verkleinerung werden die Strukturen, welche den elektrischen Strom transportieren, nur noch wenige Atomlagen betragen. Die Ladung (0/1) muss dann anhand eines einzelnen Atoms oder Elektrons unterschieden werden. Es ist abzusehen, dass es dem Transistor in einigen Jahren wie einst der Vakuumröhre ergehen wird. Ein Nachfolger könnte Hewlett Packard's Crossbar Latch sein. Weiterführende Informationen zu Crossbar Latch können in [Maxp09; Gepp05; MoJB07] nachgelesen werden.

Basierend auf der technischen Entwicklung der GPU wird im nachfolgenden Abschnitt das GPU-Programmiermodell ausführlicher untersucht.

## 2.4 GPU-Programmiermodell

*General Purpose Computation on Graphics Processing Unit (GPGPU)* bezeichnet ganz allgemein die Möglichkeit der Nutzung der GPU für allgemeine Berechnungen, sei es im semi-professionellen Umfeld oder im Anwendungsbereich des HPC. Diese Thematik und deren Ursprung in der Grafikwelt ist sehr Umfangreich und geht über den in der Arbeit geforderten Umfang hinaus, sodass hier nur auf das wichtigste eingegangen wird.

Prinzipiell ist GPGPU seit der 4. GPU-Generation (siehe Abschnitt 2.3.2) für allgemeine Berechnungen anwendbar. Allerdings musste man ein Experte der Grafikprogrammierung sein und mittels DirectX 9, *Open Graphics Library (OpenGL)* oder Shader-Hochsprachen, wie z. B. *C for graphics (Cg)*, die mathematischen

Probleme aufwendig in Shader-Programme verpacken [FeKi03, 9ff.]. Unter dem Begriff *Shader* werden hierbei Recheneinheiten innerhalb einer GPU verstanden (siehe Abschnitt 2.4.1).

An dieser Stelle knüpfte BrookGPU, ein Studienprojekt der Stanford-Universität in den USA, an und erweiterte den Brook-Compiler um einige spezielle Funktionen und ließ die GPU vergleichsweise eines Coprozessors erscheinen. Mittels des C-ähnlichen Dialekts konnte so ein Strom von Daten an einen Shader-Algorithmus gesendet werden. Daraus ergibt sich auch der Name *Stream Computing*.

Nachdem BrookGPU schließlich Interesse von ATi und nVIDIA erlangte, wechselte ein Teil der Entwickler zu nVIDIA und entwarf die Programmierschnittstelle *Compute Unified Device Architecture (CUDA)*. Der andere Teil entwarf eine für ATi-Hardware angepasste Sprache *Brook+ Stream Computing* [Bert09b]. Beide Architekturen wurden in Koordination zu der DirectX 10 API und dem Unified Shader (Shader-Modell 4) entworfen. Alte Shader-Modelle sind nicht automatisch aufwärtskompatibel, sodass für CUDA und Brook+ eine GPU ab der 5. Generation benötigt wird (siehe Abschnitt 2.3.2).

Im nachfolgenden Abschnitt wird die eingeführte Unified Shader Architektur näher erläutert.

### 2.4.1 Unified Shader

Die Recheneinheiten innerhalb der GPU werden als *Shader* bezeichnet. Der Pixel-Shader z. B. verändert Bildpunkte und der Vertex-Shader ist für das Hinzufügen von visuellen Effekten in Echtzeit, wie z. B. Regen, Nebel oder Rauch, verantwortlich. In DirectX 10 wurde zusätzlich der Geometrie-Shader für noch flexiblere Effekte spezifiziert.

Die Shader-Einheiten sind keine voneinander unabhängig arbeitende Prozessoren, sondern als Teil einer Kette – der Grafik-Pipeline – hintereinander angeordnet (Abbildung 2-17) [FeKi03, 9ff.].

Üblicherweise werden alle Oberflächen in der virtuellen 3D-Welt aus Mashes (Dreiecken) gebildet, mit einer Textur überzogen und auf dem Bildschirmraster abgebildet (vgl. Abbildung 2-17). Das Endergebnis jeder Bildberechnung sind Pixel, also Bildpunkte auf dem Monitor. Analog dazu steht der Begriff *Texel* für einen Bildpunkt in der Textur.

Bei großen Dreiecken hat der Vertex-Prozessor so gut wie nichts zu tun, aber der Pixel-Prozessor ist vollkommen beschäftigt. Bei kleinen Dreiecken ist es genau anderes herum. Da es bei dieser Granularität schwer ist ein gesundes Mittelmaß zu finden,

wurde mit DirectX 10 der *Unified Shader* eingeführt. Seitens der Hardware existiert nur noch der Unified Shader, was zu Deutsch etwa „Vereinigter Shader“ bedeutet. Der Treiber kann selbst entscheiden in welchem Modus der Shader, also die gesamte Recheneinheit, arbeitet.

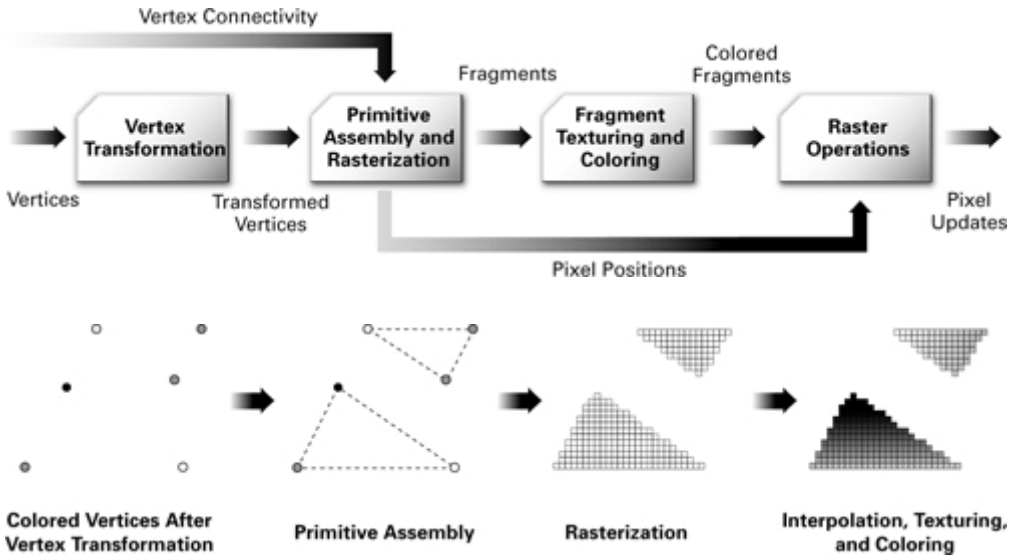



**Abbildung 2-17:** Grafikhardware Pipeline und Virtualisierung [FeKi03, 13]

Derzeit arbeiten Spiele- und Grafikchiphersteller mit Hochdruck an dem Ziel des fotorealistischen Renderings in Echtzeit. Es ist durchaus möglich, dass sich dann auch die Mashes-Technik als obsolet herausstellt und man wieder zu Voxeln (Klötzchen) zurückkehrt [Bert09a].

Nach dem kurzen Einblick in das Unified Shader-Model wird nachfolgend überwiegend auf die nVIDIA CUDA-Hardwarearchitektur sowie das nVIDIA CUDA-Programmiermodell eingegangen. Grund hierfür ist die Verwendung einer nVIDIA Grafikkarte für die Berechnung des Neuronalen Netzes (siehe Abschnitt 3.3).

### 2.4.2 nVIDIA CUDA-Hardwarearchitektur

Trotz des gemeinsamen Ursprungs von nVIDIA CUDA und ATi Brook+ sind beide Sprachen speziell auf die jeweilige Hardware angepasst und leider nicht kompatibel zueinander. Jedoch lassen sich bei Kenntnis einer Architektur schnell Analogien zu der anderen feststellen.

Die Recheneinheit (bzw. die in Abschnitt 2.4.1 verwendete Terminologie Unified Shader) der aktuellen nVIDIA High-End GPU mit der Typenbezeichnung GT200b besteht aus zehn *Texture Prozessor Cluster (TPC)*. Jeder TPC beinhaltet dabei eine Texture Unit und drei *Streaming Multiprozessoren (SM)*. In jedem SM arbeiten

letztendlich acht skalare *Stream Prozessoren (SP)*. Skalar bedeutet hierbei eine Operation pro Takt. Diese acht SP haben Zugriff 16 KB read/write Shared Memory und den read/only Constant Cache. Des Weiteren beinhaltet jeder SM zwei *Special Function Units (SFU)* für transzendente Funktionen. Transzendente Funktionen sind nicht algebraische Funktionen wie Exponential-, Logarithmus-, Sinusfunktionen usw. Erstmals neu in der GT200b-Serie ist der double precision (doppelte Genauigkeit) Prozessor. Mit älterer Grafikhardware der 5. GPU-Generation (siehe Tabelle 2-1 und Tabelle 3-4) ist es nicht möglich double precision Berechnungen durchzuführen. Die `float`-Darstellung und Operationen sind nach IEEE 754 [Ieee08] spezifiziert. Nach Adam Ries ergibt diese Konstellation in Summe 30 SM, 240 SP und 30 Double Einheiten [LNOM08; Nvid09a, 101ff.]. In Anlehnung an [LNOM08; Nvid08, 10ff.] veranschaulicht die nachfolgende Abbildung 2-18 diese Konstellation.

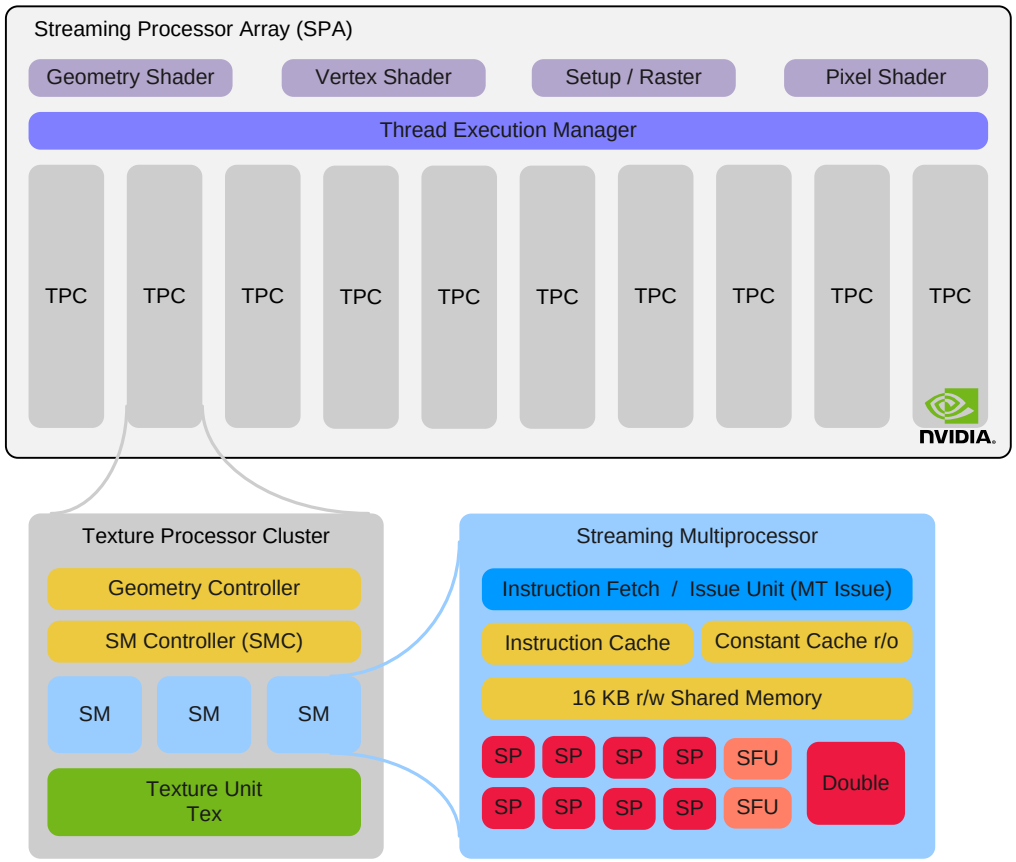



**Abbildung 2-18:** nVIDIA CUDA-Hardwarearchitektur [LNOM08; Nvid08, 10]

Die Anzahl der TPC und Anzahl der SM pro TPC sind Ausstattungs- und Leistungsmerkmale und variieren nach erworbener Leistungsvariante. In zukünftigen Serien ist mit einer Erhöhung der double precision-Einheiten zu rechnen.

Im Vergleich zu nVIDIA besitzt die aktuelle ATi GPU RV790 auf der untersten Ebene ein Pool von 160 5D Vektor SP [Ati09]. Diese können gleichzeitig fünf Operationen

abarbeiten, sofern diese voneinander unabhängig sind. Im Idealfall verhalten sie sich dann theoretisch wie 800 skalare SP von nVIDIA. Um effizient arbeiten zu können setzt die Vektorarchitektur in der Praxis große Anforderungen an den Compiler und Thread Scheduler. Unabhängige 3D-Benchmarks zeigen, dass je nach Anwendungsfall mal der eine, mal der andere Hersteller leicht die Nase vorn hat. In Verkaufsprospekten werden ATi Grafikkarten in der Regel mit 800 SP beworben. Hier sollte man sich nicht täuschen lassen. In Anlehnung an [Ati09, 1-1] veranschaulicht die nachfolgende Abbildung 2-19 die ATi R700 Hardwarearchitektur.

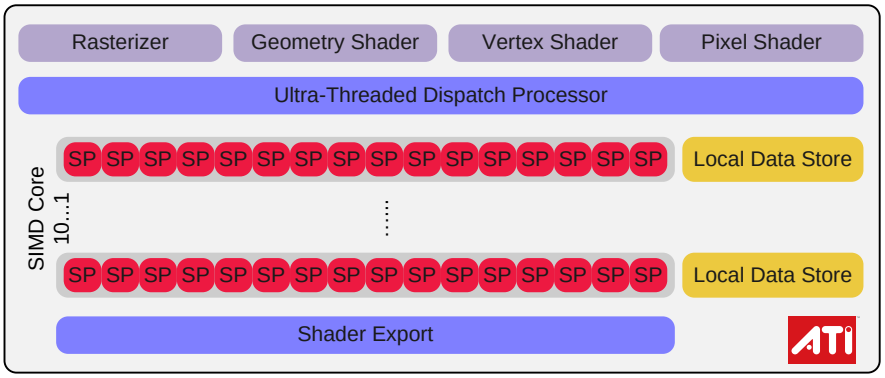



**Abbildung 2-19:** ATi R700 Hardwarearchitektur [Ati09, 1-1]

Für die Hochsprachen CUDA und Brook+ wird keine weitere Kenntnis über die Funktionsweise der Hardware vorausgesetzt. Interessierte finden ergänzende Erläuterungen der jeweiligen Hardwarearchitektur auf den Herstellerwebseiten.

### 2.4.3 nVIDIA CUDA-Programmiermodell

Der Programmierer schreibt seriellen Standard C CPU-Code, welcher *Kernel* aufruft. Kernel sind Funktionen, die auf der GPU ausgeführt werden. Abbildung 2-20 verdeutlicht die sequenziellen Kernel-Aufrufe, welche sich von der Syntax kaum von normalen Funktionsaufrufen unterscheiden.

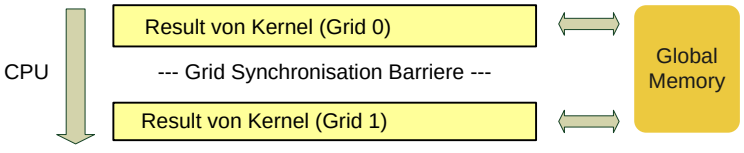



**Abbildung 2-20:** CPU Execution Modell

Der Inhalt des Kernel ist vergleichbar mit der „inner loop“ einer klassischen sequenziellen Funktion. Ein Kernel enthält jedoch parallelen Programmcode, welcher als Set von Threads auf der GPU ausgeführt wird. Jeder einzelne Thread führt immer

denselben Code des Kernel aus. Der eigentliche Thread Prozessor ist der SP-Core innerhalb des SM (vgl. Abbildung 2-18). Alle SP führen zeitgleich ein und dieselbe Operation auf unterschiedliche Daten aus. Dieses Verfahren nennt man *Single Instruction Multiple Data (SIMD)*. Im Gegensatz dazu arbeiten Mehrkern-CPU nach dem *Multiple Instruction Multiple Data (MIMD)*-Prinzip.

Bei CUDA ordnet der Programmierer die Threads hierarchisch in *Thread Blocks* innerhalb eines *Grid* an. Abbildung 2-21 verdeutlicht dieses theoretische Konstrukt.

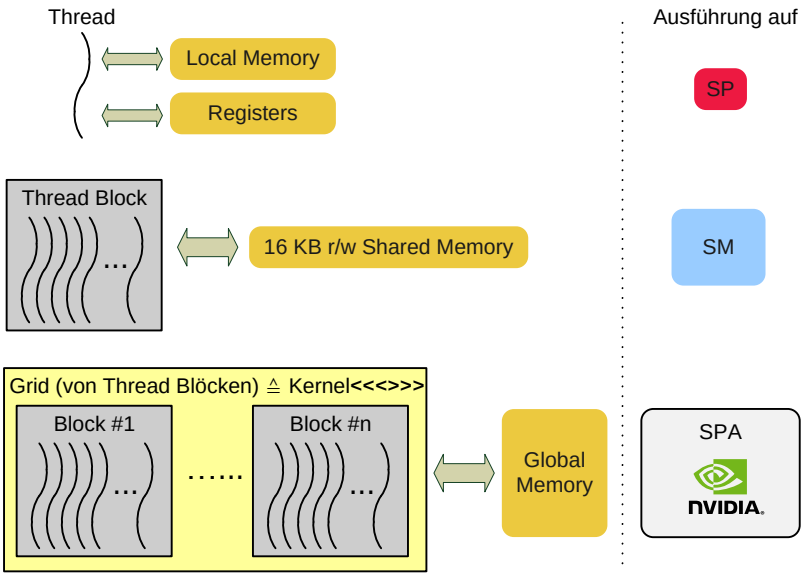



**Abbildung 2-21:** nVIDIA CUDA-Execution Modell [Nvid09c, 15]

Seitens der Syntax ist dies eine Anweisung: `<<<nBlocks, nThreads>>>` – Aufruf des Kernel mit n Thread Blocks, je n Threads (siehe Listing 2-1). Die Anzahl der Blöcke und Threads per Block muss zuvor statisch oder dynamisch – dem Anwendungsfall entsprechend – ermittelt werden. Nur Threads innerhalb eines Blockes können miteinander kommunizieren und synchronisiert werden. Eine Inter-Block Kommunikation ist hierbei nicht möglich.

Als kleines Beispiel zeigt das Listing 2-1 die Addition einer Matrix mit einem Skalar. Wie zu erkennen, wurde die „inner loop“ (`for`-Schleife) parallelisiert. Daher kommt der Kernel gänzlich ohne Schleifen aus. Der Kernel wird als Set von Threads auf der GPU ausgeführt! Sofern die Matrix eine dynamische Größe besitzt, muss die Anzahl der Threads pro Block `nThreads` und Anzahl der Blöcke `nBlocks` vor dem Aufruf des Kernel anhand der Matrizengröße ermittelt werden.

**Listing 2-1:** Vergleich CPU- und nVIDIA CUDA-Programmiermodell

```
01 // CPU (sequenziell)                  // GPU (parallel)
02
03 void inc_cpu                          __global__ void inc_gpu  // Kernel
04    (float *a,float b,int N)                    (float *a,float b,int N)
05 {                                     {
06   int idx;                             int idx=blockIdx.x * blockDim.x +
07                                                          threadIdx.x;
08   for (idx=0; idx<N; idx++)            if (idx<N)
09     a[idx] = a[idx] + b;                 a[idx] = a[idx] + b;
10 }                                     }
11
12 void main()                           void main()
13 {                                     {
14   //...                                //...
15   inc_cpu(a,b,N);                      inc_gpu<<<nBlocks,nThreads>>>(a,b,N);
16 }                                     }
```

Ein Thread Block, teils auch *Cooperative Thread Array (CTA)* genannt, kann derzeit bis zu 1.024 Threads beinhalten. Jeder Thread Block ist voneinander unabhängig und wird bis zur Beendigung auf einem SM ausgeführt. Existieren mehr Blöcke als freie Ressourcen, so werden diese nacheinander abgearbeitet. Dies hat zur Folge, dass die Reihenfolge der Abarbeitung nicht deterministisch ist! Threads innerhalb eines Blockes haben Zugriff auf den schnellen, gemeinsamen Shared Memory. Die Anzahl von Blöcken innerhalb eines Grid ist derzeit auf 65.535 begrenzt. Für das oben genannte Beispiel bedeutet dies, dass der Kernel selbst CPU-seitig mittels Schleife aufgerufen werden muss, wenn `nBlocks` > 65.535 ist. Jeder Block und jeder Thread ist über einen eindeutigen Index referenzierbar.

Damit sichergestellt ist, dass der vom Compiler erzeugte *Parallel Thread Execution (PTX)* Bytecode sowohl auf alter als auch auf zukünftiger Hardware läuft, übernimmt die CUDA-Runtime die vollständige Zerlegung und die Kompilierung des finalen PTX-Codes für die jeweilige Hardware zur Laufzeit. In Anlehnung an [Bert09b, 146] zeigt Abbildung 2-22 die Schichten von CUDA.

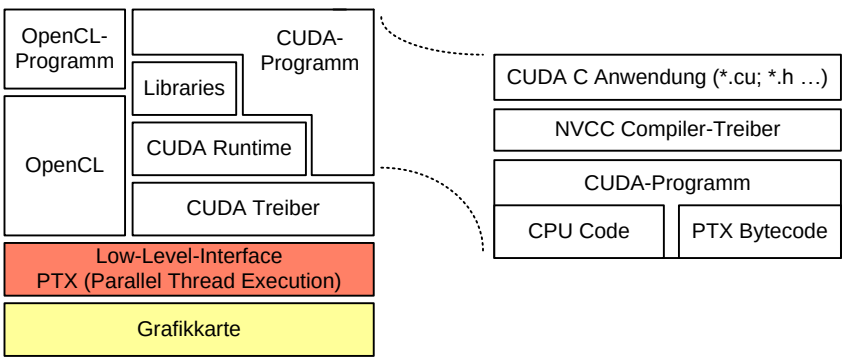



**Abbildung 2-22:** nVIDIA CUDA-Schichten [Bert09b, 146]

Der Programmierer hat keinerlei Einfluss auf die Zuordnung der Blöcke auf SM oder Threads auf SP. Für die SIMD-Architektur ist dieses auch nicht von Belang. Zu jeder Zeit kann immer nur ein Kernel auf der Grafikhardware aktiv sein. Mehrere parallele Kernel erfordern mehrere Grafikkarten. Innerhalb eines Kernel besteht kein Zugriff auf den Arbeitsspeicher des Hosts. In der Parallel-Programmierung wird der Lesezugriff einer Variable als *Gather* und der Schreibzugriff als *Scatter* bezeichnet.

Für das Grundverständnis genügen die hier dargestellten Ausführungen. Wer sich jedoch näher mit der Hardware beschäftigt, wird irgendwann auf den Begriff Warp stoßen. Die SM-Hardware ist selbst „gethreaded“. Jeder SM gruppiert intern 32 Threads zu einem Warp und kann derzeit 32 solcher Warp, also 1.024 Threads, gleichzeitig verwalten. Mit 30 SM kann die Grafikhardware zu jedem Zeitpunkt gerade mit 30.720 Threads beschäftigt sein [LNOM08; Nvid09a, 7ff.]. Ein Warp bedeutet derzeit also 32 Threads und ein Half-Warp 16 Threads. Bei der Programmierung und Optimierung ist diese interne Gruppierung teilweise von Bedeutung.

Im Ergebnis ist festzuhalten, dass ein Neuronales Netz aus Neuronen mit der vereinfachten Funktionsweise Summation Eingabe – Transferfunktion – Ausgabe besteht. Ebenso wurde die allgemeine Funktionsweise der Grafikkarte, mit vertieftem Blick auf die Komponenten Grafikspeicher, RAM-DAC und TMDS erläutert. Das Herzstück einer Grafikkarte ist die GPU. Die aktuellen GPU oder IGP der führenden Hersteller, welche den Markt quasi unter sich aufteilen (Intel 50 %, nVIDIA und ATi je knapp ein Drittel Marktanteil), besitzen bis zu 1.400 Transistoren. Die aktuelle Grafikhardware funktioniert nach dem SIMD-Prinzip, besitzt bis zu 240 Cores und führt parallele Funktionen, genannt Kernel, aus.

Im nachfolgenden Kapitel 3 wird nun auf Basis, der in diesem Kapitel erörterten Grundlagen, der Entwurf des Künstlichen Neuronalen Netzes skizziert.

# 3 Entwurf

Im folgenden Kapitel wird der Entwurf modelliert. Hierzu wird in Abschnitt 3.1 zunächst die Zieldefinition konkretisiert. Darauf aufbauend wird in Abschnitt 3.2 der Netzentwurf erarbeitet und die Parallelisierbarkeit von Neuronalen Netzen betrachtet. Abschnitt 3.3 beschäftigt sich mit Überlegungen zur Komponentenauswahl.

## 3.1 Zieldefinition

Aufbauend auf den in Kapitel 2 erläuterten theoretischen Grundlagen wird im Folgenden die Zielsetzung (vgl. Abschnitt 1.2) an die praktischen Gegebenheiten angepasst.

Wie in Abschnitt 1.1 beschrieben, werden in einem Kampfflugzeug unter anderem eigens entwickelte VME oder cPCI Boards eingesetzt. Allerdings geht der Trend auch dazu über *Commercial Off-the-Shelf (COTS)*-Komponenten, also bereits serienfertige Produkte, zu benutzen. Wird eine COTS-Komponente modifiziert, nennt man dies *Modified Off-the-Shelf (MOTS).* Die COTS- oder MOTS-Komponente wird anschließend gehärtet, da durch die enormen G-Kräfte unbehandelte Hardware bereits beim Start des Kampfflugzeug auseinander brechen würde Die entstandene *Ruggedized Off-the-Shelf (ROTS)*-Komponente ist nun für den flugtauglichen Einsatz geeignet. COTS-Komponenten können zu einer Kostensenkung und schnelleren Entwicklungszeiten beitragen, birgen jedoch auch Risiken wie z. B. höhere Integrationskosten, kostspielige Änderungen, nicht vorhandene oder nicht benötigte Funktionalitäten, nicht Haftbarkeit des Herstellers für Fehler.

Aus dem Marktüberblick (siehe Anhang D) geht hervor, dass derzeit keine COTS-Grafikboards mit GPGPU-Funktionalität für VME- oder cPCI-Systeme erhältlich sind. Vorzufinden sind lediglich ältere, kaum noch auf dem Konsumentenmarkt erhältliche Grafik-Chipsätze diverser Hersteller von der Pre-GPU bis hin zur 4. GPU-Generation. Intel fertigt zwar cPCI Boards mit IGP der 5. Generation (siehe Tabelle 2-1), stellt allerdings kein *Software Development Kit (SDK)* für die Hardware zur Verfügung. Die Präferenzen von VME- oder cPCI-Grafikboards liegen deutlich in der Zuverlässigkeit statt Performance. Der Anwendungsbereich von GPGPU beschränkt sich derzeit somit auf den Desktop- und Servereinsatz mit einer Grafikkarte bzw. einer GPU der 5. Generation von ATi oder nVIDIA.

Wie in Abschnitt 2.1.2 erwähnt, liegt die rechenaufwendige Phase in der Lernzeit des Neuronalen Netzes, welche allerdings nur einmalig am Boden durchgeführt werden muss. Die letztendliche Berechnung der Kollisionswahrscheinlichkeit im Kampfflugzeug erfordert keine besonders hohe Rechenleistung, weshalb für diesen

Anwendungsbereich kein Hochleistungsgrafikboard benötigt wird. Eine bereits vorhandene Netzbeschreibung und verschiedene Simulationsfunktionen zur Beurteilung von Kollisionen sind allerdings nicht öffentlich zugänglich. Aus diesem Grund kann im Rahmen der Arbeit lediglich prototypenhaft untersucht werden, inwieweit Grafikkarten überhaupt zur Beschleunigung des Lernens beitragen können.

Hierzu wird im folgenden Abschnitt ein Neuronales Netz für den Test erarbeitet. Nach der Auswahl der Hardware erfolgt die spezielle Implementierung auf die Hardware. Das Programm muss auf der CPU und der GPU lauffähig sein, sodass ein Leistungsvergleich für diesen Anwendungsfall erfolgen kann.

## 3.2 Netzentwurf

In diesem Abschnitt wird das Neuronale Netz präzisiert, welches zur Beurteilung der Leistungsfähigkeit einer GPU beitragen soll.

Das für die Leistungsbewertung zu Grunde liegende Neuronale Netz wurde so abgewandelt, dass lediglich ein indirekter Praxisbezug besteht. Mittels eines single layer neuronal network (siehe Abschnitt 2.1.2) soll ein optimaler Radius *r* innerhalb eines „Kollisionsfeldes“ erlernt werden.

In der Praxis besteht das Neuronale Netz aus mehreren Knoten (Neuronen) und besitzt bis zu zehn Eingabeparametern mit je zehn Eingabewerten je Parameter. Parameter sind hier z. B. Fluggeschwindigkeit oder Flughöhe des eigenen und des fremden Flugzeuges. Mit einer ähnlichen Datenmenge soll deshalb auch im Testprogramm gearbeitet werden.

Die folgende Abbildung 3-1 stellt das Neuronale Netz für die Leistungsbewertung schematisch dar. Gearbeitet werden soll vorerst mit bis zu zehn Eingabeparametern $x_0 \ldots x_9$.

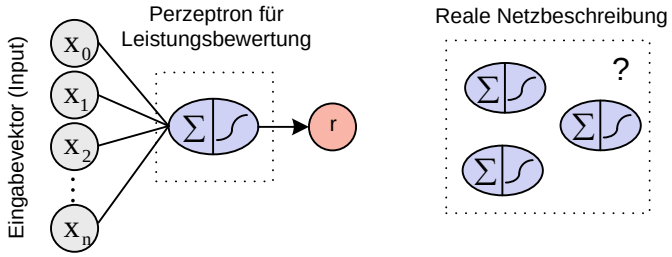



**Abbildung 3-1:** Netzbeschreibung

Für das zu implementierende Neuronale Netz spiegeln die Eingabevektoren $x_i$ die Anzahl der Dimensionen wieder. So lernt das Perzeptron im Falle von zwei Dimensionen einen Kreis und im Falle von drei Dimensionen (Raum) eine Kugel. Für mehr als drei Dimensionen ist das Resultat der Phantasie des Lesers überlassen.

Folgende Abbildung 3-2 stellt ein mögliches Kollisionsfeld mit zwei Eingabeparametern schematisch dar. Die konstanten Parameter $c_i$ stellen dabei den Mittelpunkt bzw. den Startpunkt dar.

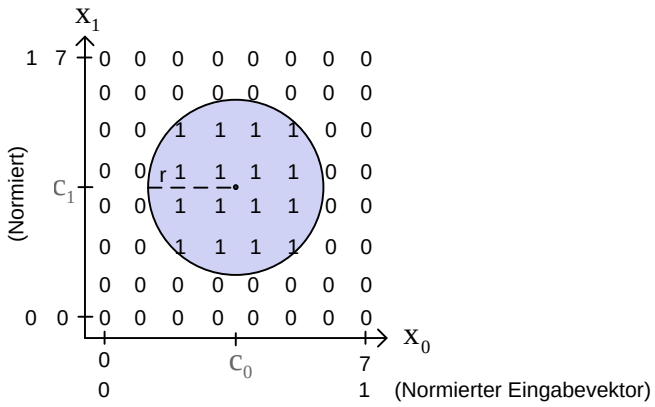


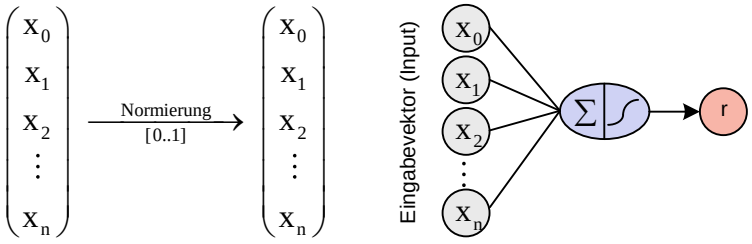

**Abbildung 3-2:** Beispiel-Kollisionsfeld mit zwei Eingabeparametern und Radius

Normalerweise stammen die Sollwerte (Testwerte) aus einer Simulationsfunktion oder einer Datenbank. Im Testprogramm werden hingegen feste Bereiche festgelegt. Die Sollwerte für das Training des Neuronalen Netzes sind in einem Feld – nachfolgend weiterhin „Kollisionsfeld“ genannt – zwischengespeichert. Weil die Eingangsmuster und Zielgrößen in dem vorliegenden Fall vorhanden sind, ist dies ein überwachtes Lernen (siehe Abschnitt 2.1.2). Aus Performance-Gründen wird dieses Kollisionsfeld im Arbeitsspeicher gehalten. Das Feld muss vor der Berechnung mittels GPU in den Grafikspeicher kopiert werden, da ein Kernel (siehe Abschnitt 2.4.3) keinen Zugriff auf den Hauptspeicher oder Hostfunktionen hat. Die Entropie eines Elementes im Kollisionsfeld beträgt 1 bit. Dies bedeutet Kollision „ja“ oder „nein“. Bei geforderten $10^{10}$ Elementen ist daher nur eine bitweise Speicherung sinnvoll. Hierbei ergibt sich ein Speicherplatzbedarf von zirka 1,16 GB. Dies wird in der folgenden Berechnung (3.1) gezeigt.

$$
\begin{aligned}
10^{10}\ \text{Elemente} \cdot 1\ \text{Byte} &\approx 9{,}31\ \text{GB} && (\text{Datentyp: } \texttt{char}) \\
10^{10}\ \text{Elemente} \cdot 2\ \text{Byte} &\approx 18{,}62\ \text{GB} && (\text{Datentyp: } \texttt{short int}) \\
\\
10^{10}\ \text{bit} &= \frac{10000000000\ \text{bit}}{8 \cdot 1024 \cdot 1024} \approx 1{,}16\ \text{GB} && (\text{bitweise Speicherung})
\end{aligned}
\tag{3.1}
$$

Wie bereits in Abschnitt 2.1.2 beschrieben, funktioniert ein Perzeptron prinzipiell: nach folgendem Schema: Eingabe → ∑ | Threshold (Aktivierungsfunktion) → Ausgabe. Wie in Beschreibung (3.2) zu sehen, werden die Eingabevektoren auf den Wertebereich

[0…1] normiert, sodass innerhalb des Neuronalen Netzes mit `float`-Darstellung gearbeitet wird.

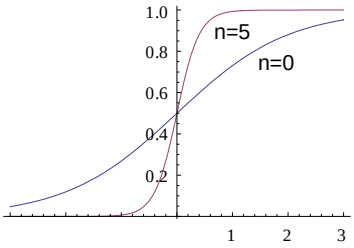



(3.2)

Zusammenfassend ergibt sich für das zu implementierende Neuronale Netz mit Summierfunktion, Ausgabefunktion und Lernregeln folgende mathematische Betrachtung (3.3).

$\sum$-Funktion: $f(x) = -(x_0 - c_0)^2 - (x_1 - c_1)^2 - \ldots - (x_n - c_n)^2 + w$ (3.3)

Aktivierungsfunktion: $A = \frac{1}{1 - e^{-n \cdot f(x)}}$ (Fermifunktion, Sigmoide)




Anmerkung:

$\lim_{x \to \infty} \triangleq$ binäre Schaltfunktion

<u>Lernregeln</u>:

Fehlerfunktion: $F = \left| Sollwert - A \right|^2$

Ableitung: $\frac{\partial F}{\partial w} = \frac{\partial F}{\partial A} \cdot \frac{\partial f}{\partial w} = \frac{\partial F}{\partial A} \cdot \frac{\partial A}{\partial f} \cdot \frac{\partial f}{\partial w} = 2 \cdot \left| Sollwert - f \right| \cdot f \cdot (1 - f) \cdot w$

Gewichtsanpassung: $\sum_{Sollwert} \Delta w = -\eta \cdot \sum_{Sollwert} \frac{\partial F}{\partial w}$

$\eta$: Lernrate

**Parallelisierbarkeit von Neuronalen Netzen**

Bevor mit der Implementierung des Neuronalen Netzes begonnen wird, ist es sinnvoll zuerst theoretisch zu prüfen, ob sich das Training eines Neuronales Netzes in voneinander unabhängige Teilstücke zerlegen lässt.

Nach vorherrschender Meinung lassen sich Neuronale Netze ideal parallel berechnen. Durch den recht einfachen Aufbau der Neuronen und einer Vielzahl gerichteter und

gewichteter Verbindungen erscheint es naheliegend das Neuronale Netz selbst auf einem verteilten System oder einem Parallelrechner abbilden und berechnen zu lassen. Dies wird von der Tatsache, dass Biologische Neuronale Netze hochgradig parallel sind, bestätigt.

Die möglichen Arten der Parallelisierung von Neuronalen Netzen [Zell03, 432f.] sind in Tabelle 3-1 zusammenfassend beschrieben.

**Tabelle 3-1:** Arten von Parallelität in Neuronalen Netzen

| | |
|---|---|
| Knotenparallelität | Die Aktivierungen aller Zellen einer Schicht werden parallel berechnet, d.h. die Knoten des Netzes werden auf verschiedene Prozessoren abgebildet. |
| Kantenparallelität | Es werden die Kanten zwischen zwei Schichten parallel berechnet, d.h. es werden die Kanten auf verschiedene Prozessoren abgebildet. |
| Trainingsmusterparallelität | Die Trainingsmuster werden hierbei parallel trainiert. Jeder Prozessor arbeitet mit dem gesamten Netz. Es werden lokale Gewichtsänderungen berechnet, welche dann entsprechend aufsummiert werden. |
| Ebenenparallelität | Die Aktivierung der einzelnen Stufen des Netzes werden parallel berechnet. |
| Phasenparallelität | Bei Backpropagation kann die Berechnung der Vorwärts- und Rückwärtspropagation parallel durchgeführt werden. In Zusammenarbeit mit Ebenenparallelität erhält man eine Art Pipelining. |
| Trainingsparameterparallelität | Das Netzwerk kann parallel mit verschiedenen Sätzen von Trainingsparametern trainiert werden. Sequenziell entspricht dies dem Starten verschiedener Trainingsläufe mittels batch. Es handelt sich also nicht um eine Parallelität im engeren Sinne, da die Netzberechnung nicht beschleunigt wird. Für Parameteroptimierung kann es dennoch interessant sein. |
| Netztopologieparallelität | Verschiedenartig Neuronale Netze werden parallel trainiert. |
| Lernverfahrenparallelität | Ein Netz wird mit verschiedenen Lernverfahren parallel trainiert. |
| Teilnetzparallelität | Teilnetze aus komplexen Netzwerken werden parallel trainiert; ggf. sogar mit unterschiedlichen Lernverfahren (Lernverfahrenparallelität). |

Die drei zuletzt beschriebenen Verfahren Netztopologieparallelität, Lernverfahren-parallelität und Teilnetzparallelität lassen sich auf SIMD-Hardware (siehe Abschnitt 2.4.3) nicht effizient durchführen.

Trainingsmusterparallelität bietet den höchst möglichen Parallelitätsgrad, gefolgt von Kantenparallelität, Knotenparallelität, Ebenenparallelität und Phasenparallelität. Höhere Parallelitätsgrade können durch Kombination der Ansätze erreicht werden [Zell03, 432f.].

Für das Testprogramm, welches nur ein einstufiges Neuronales Netz darstellt, wird Trainingsmusterparallelität implementiert. Jeder Prozessor (SIMD-Core bei ATi, Streaming Muliprozessor bei CUDA) hält dabei eine Kopie des Netzwerkes und aller Gewichte. Sollte der lokale Speicher des Prozessors nicht ausreichen, können Netzwerk und Gewichte auch global gespeichert werden, was jedoch die Broadcast-Kommunikation erhöht. Trainiert wird immer eine kleine Gruppe (Teilmenge) von Trainingsmustern. Die lokal berechneten Gewichtsänderungen werden über alle

Prozessoren aufsummiert und wieder an alle Prozessoren verteilt. Außer zur Bildung der Summe ist keinerlei Kommunikation der Prozessoren untereinander notwendig. Auf die Laufzeit nachteilig auswirken kann sich, dass sich die vielen kleinen Gewichtsänderungen oftmals gegenseitig aufheben. Das Verfahren ist nur sinnvoll, wenn eine Vielzahl von Mustern trainiert werden soll (Anzahl Trainingsmuster ≥ Anzahl Prozessoren) [Zell03, 441f.].

Die nachfolgende Abbildung 3-3 verdeutlicht die Gewichtssummierung bei Trainingsmusterparallelität.

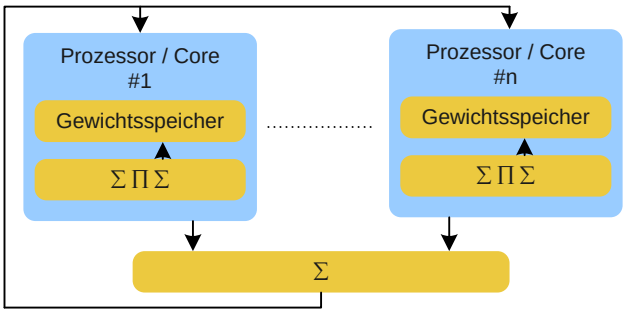



**Abbildung 3-3:** Gewichtssummierung bei Trainingsmusterparallelität

Nachfolgend wird nun die für die Implementierung relevante Komponentenauswahl näher betrachtet.

## 3.3 Komponentenauswahl

Im Mittelpunkt der Komponentenauswahl steht die Frage, welche Grafikhardware als Basis des Leistungsvergleichs angeschafft werden soll.

Im vorliegenden Fall konnte über die zu verwendende Grafikhardware frei entschieden werden. GPGPU ist mit jeder aktuell für PC erhältlichen Grafik- oder HPC-Rechenkarte von ATi oder nVIDIA möglich. Alle anderen Hersteller stellen kein SDK zur Verfügung, weshalb hier kein GPGPU möglich ist. Leider reagierte AMD (ATi) nicht auf Anfragen und bietet diese Technik dem Anschein nach nur nebenläufig an. Man erkauft sich eine Technologie ohne weiteren Support. Bei nVIDIA hingegen steht ein gutes Developer-Forum, Zugang zum Extranet und ein Ansprechpartner zur Verfügung.

Folglich wurde sich für das Prototyping für eine nVIDIA GeForce GTX 285 entschieden. Von den Spezifikationen her ist es eine der schnellsten, derzeit auf dem Markt befindlichen Grafikkarten. Sie ist nahezu identisch mit der für HPC ausgelegten nVIDIA Tesla C1060 Rechenkarte. Im Vergleich zur nVIDIA GeForce GTX 285 besitzt die nVIDIA Tesla C1060 keinen Monitorausgang und 4 GB Grafikspeicher anstelle von 1 GB (siehe Tabelle 2-2). Allerdings sprechen die GPU-Hersteller ATi und nVIDIA

keine explizite Empfehlung zur Nutzung günstigerer Standard-Grafikkarten für HPC aus, da sich die Produzenten der Grafikkarten letztendlich nicht an das von ATi oder nVIDIA vorgegebene Referenzdesign halten müssen. In diesem Zusammenhang sei der Hersteller PNY Technologies erwähnt, welcher als alleiniger Produktpartner die Tesla Serie von nVIDIA fertigt.

Sofern bei der Evaluierung eine zufriedenstellende Leistungssteigerung mit Grafikhardware feststellt werden kann, wird die Portierung und das Training des vorhandenen Neuronalen Netzes jedoch nicht adhoc durchführbar sein. Die Schnelllebigkeit des Marktes und der Sachverhalt, dass bereits Grafikkarten der 6. GPU-Generation (siehe Tabelle 2-1) angekündigt sind, rechtfertigte nicht die Anschaffung einer Tesla HPC Rechenkarte oder einer mit Tesla bestückten HPC-Workstation. Das Testergebnis der nVIDIA GeForce GTX 285 sollte sich nahezu identisch auf einer nVIDIA Tesla C1060 reproduzieren lassen.

Aufgrund von Lieferungsengpässen konnte keine PNY nVIDIA GeForce GTX 285 beschafft werden, sodass auf eine ASUS nVIDIA GeForce GTX 285 zurückgegriffen werden musste. Die Spezifikationen stimmen mit den Spezifikationen des Referenzdesigns überein (siehe Tabelle 2-2 und Tabelle 3-4).

Der Sachverhalt aus Formel (3.1) lässt erkennen, dass sich $10^{10}$ Elemente nicht im Grafikspeicher einer nVIDIA GeForce GTX 285 abbilden lassen. Ein ständiger partizieller Transfer zwischen Host und Device würde die theoretische Leistung verfälschen, sodass mit einer kleineren Datenmenge gearbeitet wird. Als Testgrundlage wurde daher ein Kollisionsfeld mit $9^9$ Elementen vereinbart.

Nachfolgend werden die relevanten Systemanforderungen, die Konfiguration des Testsystems sowie die ersichtlichen Möglichkeiten, Grenzen und Probleme näher betrachtet.

### 3.3.1 Systemanforderungen

Grundsätzlich werden keine besonderen Anforderungen an das Entwicklungssystem gestellt. Sofern kein vorkonfiguriertes GPU-System erworben wird, ergeben sich die Hard- und Softwarevoraussetzungen wie in Tabelle 3-2 dargestellt.

**Tabelle 3-2:** CUDA Systemanforderungen

| | |
|---|---|
| Hardware | · Standard-PC |
| | · PCIe Slot (16× empfohlen) |
| | · nVIDIA CUDA fähige Grafikkarte (GeForce 8 oder höher, Tesla…) |
| | · Empfohlen: Hauptspeicher > Grafikspeicher |
| | · Ausreichend dimensioniertes Netzteil!<br>Anhaltspunkt: 300W System + 250W je GPU |
| Software | · Betriebssystem Windows, Linux, MacOS (32 bit / 64 bit) |
| | · Grafikkartentreiber (beinhaltet CUDA-Runtime) |
| | · CUDA-Toolkit (beinhaltet Compiler-Treiber und Libraries) |
| | · CUDA-SDK (Development Kit beinhaltet Beispiele und Konfigurationen) |
| | · C/C++ Compiler zur Kompilierung des CPU-Anteils |
| | · Entwicklungsumgebung wie Eclipse oder Visual Studio 2005 / 2008 vorteilhaft |

Grafikkartentreiber, CUDA-Toolkit und CUDA-SDK werden vom GPU-Hersteller kostenlos zur Verfügung gestellt. Es besteht die Möglichkeit ein CUDA-Programm auch ohne entsprechende Hardware zu entwickeln. Hierfür werden lediglich das CUDA-Toolkit und das CUDA-SDK benötigt. Mit Hilfe des Emulationsmodus kann hierbei das Programm getestet werden. Der Emulationsmodus ist zudem äußerst hilfreich beim Debuggen der Anwendung.

Für die Kompilierung des CPU-Anteils innerhalb einer CUDA-Anwendung wird zusätzlich ein C/C++ x86 bzw. x86-64 Compiler benötigt. Dies kann gcc oder unter Windows Visual Studio 2005/2008 (auch kostenlose Express Edition) sein.

Ferner sollten einige wichtige Hinweise berücksichtigt werden. Empfehlenswert ist es, die für GPGPU verwendete Grafikkarte in einem PCIe 16× Slot zu installieren, um den Transfer von Host (CPU, RAM) zu Device (GPU, Grafikspeicher) so schnell wie möglich durchführen zu können. Sofern mehrere Grafikkarten (bis zu vier eines Herstellers) installiert werden, so ist auf Mainboards mit unabhängigen PCIe 16× Slot zu achten. Auch sollte das SLI in den Grafikkarten-Optionen deaktiviert werden, da miteinander verbundene Grafikkarten sich extern wie eine einzelne verhalten. Das ist bei GPGPU in der Regel nicht gewünscht.

Werden normale GeForce oder Radeon Grafikkarten anstelle von Tesla oder FireStream Karten für HPC genutzt, ist es sinnvoll – aber nicht notwendig – eine extra Grafikkarte für die Monitorausgabe bereit zu stellen. Grund hierfür ist, dass die Grafikausgabe, das Füllen des Framebuffers etc. höher priorisiert wird als eine anderweitige Nutzung.

Seit der Einführung des i386DX im Jahr 1985 dominieren die 32 bit Systeme. Gegenwärtig bieten diese Systeme noch bestmögliche Geschwindigkeit und Kompatibilität. Grundsätzlich kann ein 32 bit System nur $2^{32}\ \text{bit} = 4\ \text{GB}$ adressieren.

Bei der Installation von 4 GB oder mehr Hauptspeicher werden die von Systemgeräten genutzten Adressräume jedoch wieder abgezogen. Den größten Anteil dürfte hier die Grafikkarte in Anspruch nehmen. Mit 6 GB Hauptspeicher und einer Grafikkarte mit 512 MB Grafikspeicher bleiben als Konsequenz nur rund 3,5 GB adressierbarer Hauptspeicher. Inwieweit sich eine Tesla-Karte in einem solchen System überhaupt betreiben lässt sei dahingestellt. Allerdings besitzt Intel seit Pentium Pro und AMD seit Athlon einen 36 bit Adressbus. Hier kann man mittels *Physical Address Extension (PAE)* diesem Effekt entgegen wirken. Mit einem geeigneten Mainboard unterstützt Linux seit Kernel 2.4.4 volle 64 GB RAM (über Kernelconfig-Option einstellbar). Bei Microsoft kommt es auf die Version und den Typ des Betriebssystems an: Die Desktop-Versionen unterstützen maximal 4 GB RAM und 2 GB pro Prozess. Die Server-Varianten unterstützen je nach Version 4, 8, 32 oder 64 GB Hauptspeicher. Mit einem reinen 64 bit System können diese Probleme vermieden werden.

### 3.3.2 Testsystem

Der Test erfolgt auf einem bestehenden System. Hierfür wurde vom Administrator die bestehende Grafikkarte GeForce 8800 GTS durch die neue GeForce GTX 285 ersetzt, der Grafiktreiber aktualisiert, CUDA-Toolkit und CUDA-SDK installiert. Die Systemkonfiguration des Testsystems ist in Tabelle 3-3 dargestellt.

**Tabelle 3-3:** Systemkonfiguration des Testsystems

| | |
|---|---|
| Systemkonfiguration | • Intel Xeon (2 GHz, 8 Cores)<br>• 16 GB RAM<br>• Betriebssystem: OpenSuSE Linux 32 bit, Version 10.3<br>• CUDA-Toolkit 2.1 (aktuelle Version 2.3 nicht für OpenSuSE < 11 erhältlich)<br>• CUDA-SDK 2.1<br>• NVIDIA Treiber für OpenSuSE Linux mit CUDA Support, Version 180.22 |

Auf einem baugleichen System steht zum Vergleich weiterhin eine nVIDIA GeForce 8800 GTS zur Verfügung. In der nachfolgenden Tabelle 3-4 erfolgt eine Gegenüberstellung der beiden Grafikkarten nVIDIA GeForce 8800 GTS und GTX 285 (Daten siehe Anhang E).

**Tabelle 3-4:** Gegenüberstellung nVIDIA GeForce 8800 GTS und GTX 285

| | nVIDIA GeForce 8800 GTS | nVIDIA GeForce GTX 285 |
|---|---|---|
| Jahr der Markteinführung | 2007 | 2009 |
| GPU Modell Nr. | GT92 | GT200b |
| Anzahl Transistoren | 981 Mio. | 1400 Mio. |
| Fertigungsprozess | 90 nm | 55 nm |
| GPU Takt | 500 MHz | 648 MHz |
| RAM Takt (Größe / Typ) | 800 MHz (640 MB / GDDR3) | 1242 MHz (1024 MB / GDDR3) |
| Speicheranbindung (Theoretische Bandbreite) | 320 bit (64 GB/s) | 512 bit (159 GB/s) |
| Shader Takt | 1,19 GHz | 1,48 GHz |
| Multiprozessoren (Stream / Double Prozessoren) | 12 (96 / 0) | 30 (240 / 30) |
| DirectX / OpenGL Kompatibilität | 10 / 2.1 | 10 / 3 |
| CUDA Kompatibilität | Version 1.0 | Version 1.3 |

### 3.3.3 Ersichtliche Möglichkeiten, Grenzen und Probleme

Die Anwendungsgebiete für HPC sind weiträumig, wie z. B. Datenanalyse, Kryptografie, Simulation oder Bildrekonstruktion.

Ob eine Berechnung von der Leistung einer GPU profitieren kann, hängt von verschiedenen Faktoren ab. Es ist natürlich nicht sinnvoll eine einfache Addition von einer GPU ausführen zu lassen. Um die SIMD Recheneinheiten kontinuierlich mit einem Strom von Daten zu versorgen, benötigt es schon mindestens eine Millionen gleichartiger Berechnungen. Vor allem der Transport zwischen Hauptspeicher und Grafikspeicher stellt einen Flaschenhals dar. Um einen Geschwindigkeitsvorteil gegenüber der CPU zu erreichen, ist teilweise entscheidend, ob die Daten in den Grafikspeicher passen und dort mehrfach verwendet werden können oder nicht.

Weiterhin sollte ein Kernel (siehe Abschnitt 2.4.3) aus vielen Rechenoperationen und möglichst wenigen globalen Speicherzugriffen bestehen, da die Grafikkarte nur wenige und ineffiziente Caches besitzt [Bert09b; FiSt09].

Zusätzlich ist die benötigte Genauigkeit im Vorfeld abzuklären. Gegenwärtige Grafikhardware besitzt 8× mehr single precision- wie double precision-Einheiten. Sofern double precision unverzichtbar ist, sind Algorithmen wie mixed precision empfehlenswert. Dies ist ein Verfahren, welches zunächst mit single precision rechnet und den letzten Schritt mit double precision durchführt [FiSt09; KuDo06]. Ältere Grafikkarten, wie z. B. die GeForce 8800 GTS, besitzen keine double precision-Einheiten (siehe Abschnitt 2.4.2).

Ein anderes Problem können Treiber darstellen. Hiervon ist sicher kein Hersteller befreit, jedoch zeigt sich gerade im Massenmarkt, dass aufgrund des harten Wettbewerbs teils sog. „Bananenware“ produziert wird. Dies sind Produkte die beim

Kunden reifen. Üblicherweise existiert ein Referenztreiber für alle Grafikkartenmodelle eines Herstellers. Auch wenn Optimierungen in den meisten aller Fällen den Anwendern zu Gute kommen, kann die Kompatibilität verloren gehen. Sofern die Änderungen nicht (sicherheits-)relevant sind, ist es nicht selten günstiger nach dem Grundsatz „never change a running system“ zu handeln.

Im Vergleich zu anderer HPC-Hardware sind die Speichermodule bei Grafikkarten, aber auch bei den für HPC angebotenen Rechenkarten, Tesla und FireStream, auf Geschwindigkeit optimiert. *Error Correction Code RAM (ECC RAM)*, welcher wegen interner Fehlerkontrolle allerdings deutlich Leistungsschwächer als GDDR RAM ist, wird nicht angeboten. Bei RAM-Modulen ohne Fehlerkontrolle sind Bitfehler nicht direkt erkennbar. Eine zweieinhalb-jährige Studie von Google [ScPW09], welche in ihren Serversystemen ausschließlich ECC RAM einsetzen, ergab, dass innerhalb eines Jahres mindestens ein korrigierbarer Bitfehler bei rund ein Drittel aller Server protokolliert wurde. Dies entspricht einer Wahrscheinlichkeit von rund 8 % pro RAM-Modul. Nicht korrigierbare Bitfehler, welche zum Serverabsturz oder zum Herunterfahren des Systems führten betrugen 1,3 % pro Server bzw. 0,22 % pro RAM-Modul. Durch Umwelteinflüsse hervorgerufene Softerrors traten hierbei deutlich seltener auf, als Fehler durch Hardwaredefekte.

In diesem Kapitel wurde das Neuronale Netz mathematisch formal beschrieben und die Systemvoraussetzungen geklärt. Das Neuronale Netz bzw. der Netzknoten, das Perzeptron, soll einen Radius über mehrere Dimensionen lernen. Aufgrund der beschafften Hardware nVIDIA GeForce GTX 285 mit 1 GB Grafikspeicher besteht das Kollisionsfeld aus maximal $9^9$ Elementen statt später geforderten $10^{10}$ Elementen. Das Kollisionsfeld ist ein Speicherbereich, welches die Sollwerte (Testwerte) für das überwachte Lernen beinhaltet. Die Speicherung des Kollisionsfeldes erfolgt bitweise innerhalb von `char`-Elementen. Mittels GPU werden die Trainingsmuster parallel trainiert. Jeder der 30 SM (Streaming Multiprozessoren) errechnet dabei lokale Gewichtsänderungen, welche abschließend über alle SM aufsummiert werden. Das Kapitel schloss mit einem Überblick über die Konfiguration des Testsystems und einigen weiterführenden Hinweisen ab.

Aufbauend auf den Entwurf wird im nachfolgenden Kapitel 4 auf die Implementierung eingegangen.

# 4 Implementierung

Aufbauend auf dem Entwurf soll in diesem Kapitel die Implementierung, der für die Bewertung wichtigen Programmteile, in Auszügen erläutert werden. Das implementierte Testprogramm besteht aus den in Tabelle 4-1 beschriebenen Modulen.

**Tabelle 4-1:** Programmteile des Testprogramms CUDABench

| | Beschreibung |
|---|---|
| devices.cu<br>devices.h | Listet alle CUDA-fähigen Geräte und zeigt die Spezifikationen, wie z. B. Anzahl der Prozessoren, an. Bei mehreren CUDA fähigen Grafikkarten kann das Gerät für die Berechnung gewählt werden. |
| matrixtests.cu<br>matrixtests.h | Einfacher Matrixtest als Einführung in die parallel Programmierung und zur Überprüfung der in Abschnitt 3.3.3 aufgestellten Thesen. |
| collisionfield.cu<br>collisionfield.h | Beinhaltet alle nötigen Funktionen für das Erzeugen (und Anzeigen, Laden, Speichern) des Kollisionsfeldes. Das Kollisionsfeld ist ein logischer Bezeichner für eine Speicherstelle, wo die Sollwerte (Testwerte) für das überwachte Lernen des Neuronalen Netzes hinterlegt sind. |
| neuronalnet.cu<br>neuronalnet.h | Enthält die Routinen des Neuronalen Netzes. Ebenso ist das Rosenblatt Perzeptron (siehe Abschnitt 2.1.2) zu Illustrationszwecken enthalten. |
| main.cu | Hauptprogramm (Menu)<br>Die Module Matrixtest, Erzeugen des Kollisionsfeldes, Neuronales Netz-Training sind jeweils auf CPU und auf GPU ausführbar. |

Für die Leistungsbewertung (siehe Kapitel 5) sind jedoch nur zwei Teile von direkter Bedeutung. Das Erzeugen des Kollisionsfeldes und das Training des Neuronalen Netzes. Beide Teile können jeweils auf der CPU oder GPU ausgeführt werden. Programmiert wurde in C bzw. CUDA für den GPU-Anteil. Nachfolgend wird in Abschnitt 4.1 zunächst die Implementierung des Kollisionsfeldes betrachtet. Als nächstes wird in Abschnitt 4.2 auf die Implementierung des Neuronalen Netzes eingegangen und noch zu lösende Probleme aufgezeigt. Anschließend wird in Abschnitt 4.3 die Problematik der Programmoptimierung angesprochen.

## 4.1 Implementierung des Kollisionsfeldes

Das Kollisionsfeld, also der Speicherbereich, enthält die Sollwerte für das Training des Neuronalen Netzes (siehe Abschnitt 3.2). Normalerweise stammen die Sollwerte selbst aus einer (rechenintensiven) Simulationsfunktion oder einer Datenbank. Für das Testprogramm ist es jedoch unerheblich, ob es sich um reelle Werte handelt oder nicht. Deshalb werden beim Erzeugen des Kollisionsfeldes nur statische Bereiche vorgegeben.

Prinzipiell könnte man die Sollwerte auch in einem Array abspeichern, was jedoch aufgrund der Vielzahl von Datensätzen und dem kleinen Informationsgehalt eines Elementes (vgl. Formel (3.1)) nicht in Frage kommt. Für die bitweise Speicherung der Informationen kann man in C/C++ Bitfelder definieren. Der ANSI C Standard sieht die

Nutzung des `int` Datentyp innerhalb eines `struct` vor [Msdn08a]. Das folgende Listing 4-1 verdeutlicht die Funktionsweise.

**Listing 4-1:** C/C++ Bitfeld

```
01 struct Kollisionsfeld // Bitfeld
02 {
03   unsigned int a:16; // 16 bit
04   unsigned int b:8;  //  8 bit
05   unsigned int c:8;  //  8 bit
06 } CF[1];
07
08 void main(void)
09 {
10   CF[0].a=3;   // => 00000000 00000011
11   CF[0].b=0;   // => 00000000
12   CF[0].c=255; // => 11111111
13 }
```

Das Bitfeld aus Listing 4-1 wird wie folgt im Speicher abgelegt:

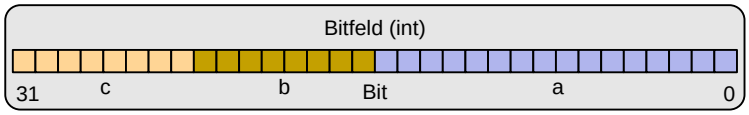



**Abbildung 4-1:** C/C++ Bitfeld

Die Speicherung des Kollisionsfeldes erfolgt allerdings nicht mittels Bitfeld, da die Referenzierung über den Namen im vorliegenden Fall eher nachteilig wirkt. Prinzipiell lässt sich auch mit standardisierten Datentypen ähnliches erreichen. Abbildung 4-2 symbolisiert hierzu die bitweise Speicherung in `char`-Elementen. Der Datentyp `char` ist mit 1 Byte (8 bit) einer der kleinsten. Der Typ `boolean` belegt Hardwaretechnisch bedingt ebenso 1 Byte. Da das Feld logisch (über Datentypgrenzen hinweg) zusammenhängend ist, referenziert die Variable `bitelement` einen theoretischen, eindeutigen Index.

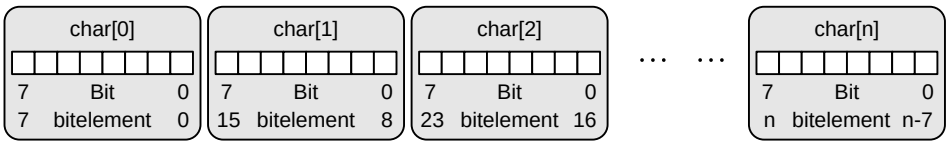



**Abbildung 4-2:** Speicherung des Kollisionsfeldes

Der Zugriff auf die unterste Ebene erfolgt über Bitoperationen. Das folgende Listing 4-2 verdeutlicht dies.

**Listing 4-2:** Initialisierung und Zugriff auf Kollisionsfeld

```
01 unsigned char* CF; // Deklaration eines Zeigers auf das Kollisionsfeld
02
03 void main(void)
04 {
05   // Definition 5 char Elemente (je 8 bit) und initialisiere Feld mit 0
06   CF=(unsigned char*) calloc(5, sizeof(char));
07
08   CF[0]|=(1<<3); // 3. bit auf 1 setzen: 00000100
09
10   printf("%i", (CF[0]&(1<<2))==0?0:1); // Testausgabe 2. bit (0)
11   printf("%i", (CF[0]&(1<<3))==0?0:1); // Testausgabe 3. bit (1)
12
13   free(CF); // Speicher freigeben
14 }
```

Im Vergleich zum Listing 4-2 fehlt beim Erzeugen des Kollisionsfeldes im Testprogramm grob vereinfacht noch eine Schleife um Zeile 8 über alle Elemente und das Setzen des Bit (Sollwert) in Abhängigkeit des Rückgabewertes einer Simulationsfunktion (`if`-Abfrage). Bei paralleler Abarbeitung muss statt der einfachen Zuweisung in Zeile 8 eine *parallele Reduktion* durchgeführt werden, da gleichzeitige Zuweisungen sich gegenseitig überschreiben würden. Da dies auch im Neuronalen Netz Anwendung findet, wird an dieser Stelle auf weiterführende Erklärungen verzichtet.

Nachfolgend wird nun auf die Implementierung des Neuronalen Netzes und die angesprochene parallele Reduktion eingegangen.

## 4.2 Implementierung des Neuronalen Netzes

Die mathematische Beschreibung (3.3) des Neuronalen Netzes soll zunächst etwas programmierfreundlicher dargestellt werden. Hierfür zeigt Abbildung 4-3 ein vereinfachtes Struktogramm für das Training und die Anwendung des Neuronalen Netzes.

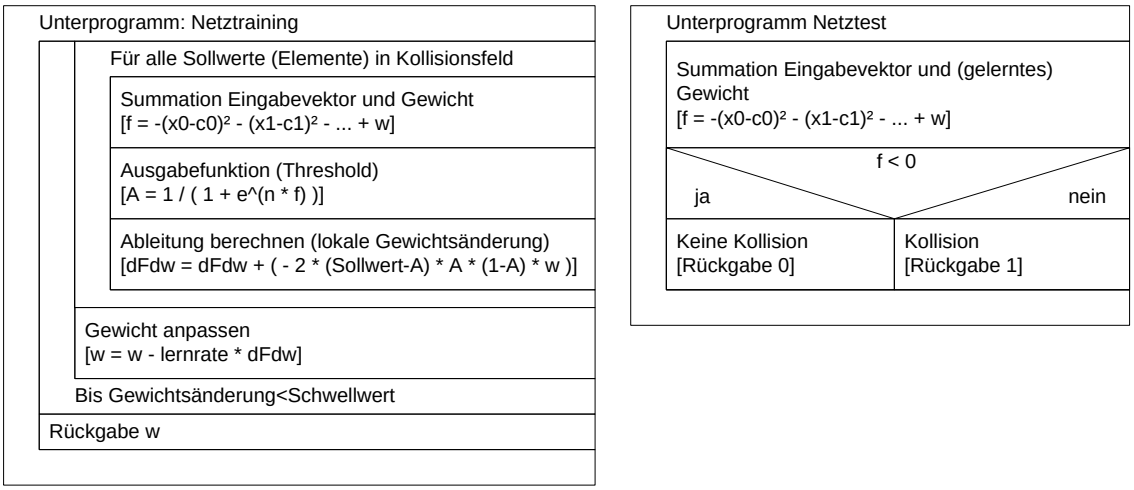



**Abbildung 4-3:** Vereinfachtes Struktogramm Netztraining und Anwendung

Aus dem Entwurf (siehe Abschnitt 3.1), Trainingsmuster parallel zu trainieren und die Laufzeit zu ermitteln, ergibt sich, dass die Ausführung der „inner loop“ – „Für alle Elemente in Kollsionsfeld“ – parallelisiert wird. Nachfolgend wird daher in Auszügen auf die parallele Umsetzung des Netztrainings eingegangen. Die sequenzielle Version ist ohne größere Schwierigkeiten aus dem Struktogramm herleitbar.

Bevor seitens der GPU das parallele Netztraining stattfinden kann, muss zunächst das Kollisionsfeld in den Grafikspeicher kopiert werden. Dieses Feld besteht aus vielen `char`-Elementen (siehe Abschnitt 4.1). Bei Trainingsmusterparallelität arbeitet hierbei jeder der 30 SM (Streaming Multiprozessoren) immer mit einem `char`-Element, die SP (Stream Prozessoren) arbeiteten auf Bit-Ebene. Die Referenzierung des `char`-Elementes erfolgt über den Index `blockIdx.x`. Das Bit (Sollwert), was logisch einem SP zugeordnet wird, wird über das Schlüsselwort `threadIdx.x` angesprochen. Da die CUDA-Runtime die vollständige Zerlegung übernimmt, lässt sich die Zuordnung der Elemente auf die jeweiligen SM nicht beeinflussen. Die nachfolgende Abbildung 4-4 stellt daher eine theoretische Betrachtung der Zuordnung dar.

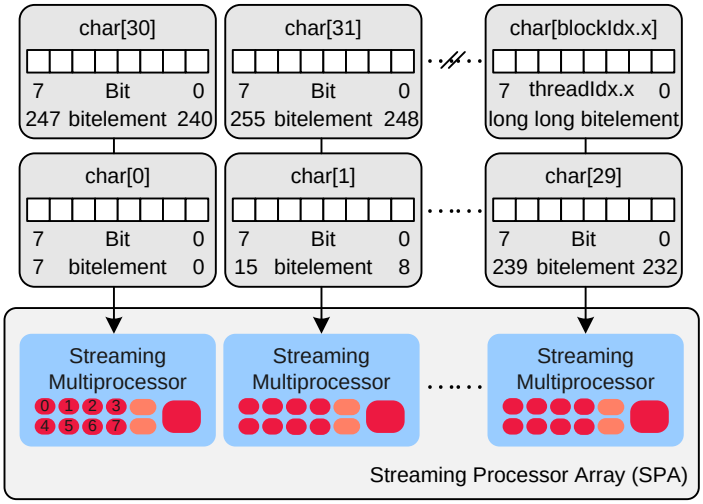



**Abbildung 4-4:** Abbildung der Sollwerte auf die Prozessoren der GPU

Jeder der acht SP innerhalb eines SM errechnet dabei eine lokale Gewichtsänderung, welche sinnvollerweise gleich mittels paralleler Reduktion SM-intern aufsummiert wird. Nachdem alle `char`-Elemente des Kollisionsfeldes durchlaufen wurden, erfolgt eine weitere Summation über die SM-Grenzen hinweg (vgl. Abbildung 3-3). Das Prinzip der parallelen Reduktion zeigt Abbildung 4-5.

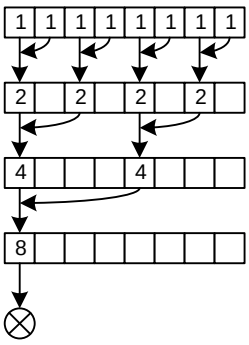



**Abbildung 4-5:** Parallele Reduktion

Nach den oben genannten Ausführungen ist die Umsetzung in die Programmiersprache zunächst nur eine Frage der Syntax. Das Listing 4-3 zeigt das Netztraining aus dem Testprogramm. Aus Gründen der Übersichtlichkeit ist dieser aus Programmteilen zusammengesetzt und hat Pseudocodecharakter. Der Quellcode wird nachfolgend erläutert.

**Listing 4-3:** Netztraining

```
__global__ void NN_Training_kernel(unsigned char* CF, float* dFdw, float w)
{
  int charelement=blockIdx.x; // char[0]...char[n]
  int bitofcharelement=threadIdx.x; // Position (bit) innerhalb des char
  int charsize=blockDim.x; // Blockgöße = char-Größe = 8 (bit)

  float f; // Funktionswert

  __shared__ float dFdwBit[8];      // Zwischenspeicher
  // extern shared float dFdwBit[]; // Siehe Anmerkung (1)

  long long bitelement=                                                ↵
        charelement*charsize+bitofcharelement; // Siehe Anmerkung (2)
  // f = -(x1-c1)² - (x2-c2)² - ... + w        // Summation Eingabevektor
  f=1/(1+expf(20*-f)); // Ausgabefunktion

  // Berechnung einer lokalen Gewichtsänderung
  dFdwBit[bitofcharelement]=-2 * ((((CF[charelement]&                   ↵
                        (1<<bitofcharelement))==0)?0:1)-f) * f * (1-f) * w;

// Parallele Reduktion
  for (unsigned int i=charsize/2; i<charsize; i*=2)
  {
    if (bitofcharelement % (2*s) == 0)
          dFdwBit[bitofcharelement]+=dFdwBit[bitofcharelement+i];
  }
  if (bitofcharelement==0)
          dFdw[charelement]+=dFdwBit[0]; // Ergebins dFdw des SM
}

void main()
{
  // Annahme: CF (Kollisionsfeld) existiert und wurde bereits "gefüllt"

  // Grafikspeicher Speicherallokation
  // Erzeugen von 500 char Elementen im Grafikspeicher
  cudaMalloc((void**) &gCF, sizeof(char)*500);
  // SM Gewichtsspeicher
  cudaMalloc((void**) &gdFdw, sizeof(float)*500);

  // Transfer der Daten von Host -> Device
  // Transfer Kollisionsfeld
  cudaMemcpy(gCF, CF, sizeof(char)*500, cudaMemcpyHostToDevice);

  do
  {
    // Kernel Aufrufen -> Grid: 500 Blöcke, je 8 Threads
    NN_Training_kernel<<<500, 8>>>(gCF,gdFdw,w); // Siehe Anmerkung (3)

    // Parallele Reduktion aller dFdw, die innerhalb des SM
    // berechnet wurden
    sum_dFdw_kernel<<<...>>>(...);
    // dFdw Device -> Host
    cudaMemcpy(&dFdw, gdFdw, sizeof(float), cudaMemcpyDeviceToHost);

    w-=learnrate*dFdw; // Gewicht ändern

  } while(w < Schwellwert);

  // Grafikspeicher bereinigen
  cudaFree(gCF);
  cudaFree(gdFdw);
}
```

Im Listing 4-3 lassen sich wichtige Sprachelemente erkennen. Kernel haben in CUDA den Funktions-Qualifier `__gobal__` (Zeile 1). Daneben bezeichnet `__device__` eine (inline) GPU-Funktion, die von einem Kernel aufgerufen werden kann. Ist kein Qualifier oder `__host__` vorangestellt, sind dies CPU-seitige Funktionen [Nvid09a, 105f.]. Die Build-In Variablen `blockIdx`, `threadIdx`, `blockDim` (Zeilen 3 bis 5) beinhalten die Indexe des aktuellen Blocks, Threads und die Größe des Blocks.

Neben Funktionen können auch Variablen Qualifier besitzen. `__shared__` spezifiziert eine Shared Memory Variable (Zeile 9). Kein Qualifier (Zeile 7) deklariert eine lokale Variable im globalen Grafikspeicher, `__global__` deklariert eine globale Variable und `__constant__` eine globale Konstante [Nvid09a, 106f.].

Zugriffe auf den (globalen) Grafikspeicher sollten minimiert und wenn möglich der Shared Memory bevorzugt werden. Auf den Speicher können acht Speicheroperationen pro Takt durchgeführt werden. Der Zugriff auf den Grafikspeicher hat jedoch eine Latenz von zirka 400 bis 600 Zyklen, was durch den Thread Sheduler ausgeglichen werden kann, sofern genügend nicht wartende arithmetische Instruktionen vorhanden sind [Nvid09b, 45].

Beim Shared Memory (siehe Abschnitt 2.4.2) ist zu beachten, dass dieser in Blöcken organisiert ist. Abbildung 4-6 stellt die blockweise Organisation des Shared Memory dar. Sofern aus einem Half-Warp (16 Threads) zwei Adressen eines Speicherzugriffes auf denselben Block fallen, entsteht ein Bank Konflikt und der Zugriff wird serialisiert. Der zweite Half-Warp eines Warp fällt in separate Instruktionen. Konfliktfrei ist ebenso, wenn alle Threads eines Half-Warp innerhalb einer 32 bit Adresse aus dem Block lesen [Fuji08, 2]. Für eine optimale Performance sollte der Zugriff konfliktfrei sein.

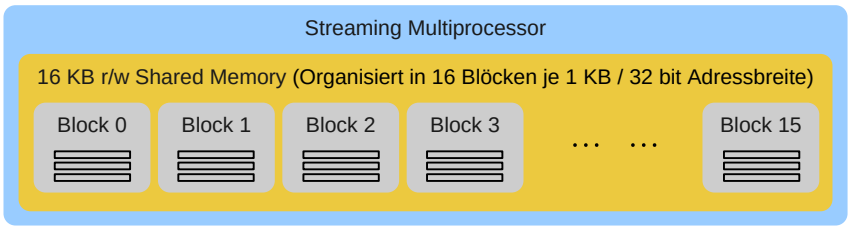



**Abbildung 4-6:** Shared Memory Bank Konflikt

Der Aufruf des Kernels erfolgt über die CUDA-Syntax `kernel<<<nBlocks,nThreads>>>(Params)` (Zeile 50). CUDA unterstützt offiziell kein C++. Einige Sachen wie z. B. Templates funktionieren, werden jedoch nicht für den produktiven Einsatz empfohlen. Für die Entwicklung hilfreich und zu Debug-Zwecken können im Emulationsmodus auch innerhalb eines Kernel Host-Funktionen aufgerufen werden. Eine der wichtigsten Host-Funktionen in diesem Zusammenhang ist `printf`.

Der Emulationsmodus ist ein Segen bei der Programmierung und zugleich ein Fluch, wenn die reale Hardware andere Ergebnisse liefert. Oftmals stimmen die Pointer nicht oder es wird ein `__syncthreads()` benötigt. Falsche Pointer fallen schwerer auf, da die Video-Hardware emuliert wird und daher Grafikspeicher und Hauptspeicher derselbe ist.

Nachfolgend wird auf die bei der Implementierung gestoßenen Probleme und auf die Anmerkungen im Quellcode näher eingegangen.

**Offene Probleme und Quellcode-Anmerkungen**

Bei der Implementierung des Neuronalen Netzes sind Probleme aufgetaucht, die im Folgenden näher erläutert werden.

*Quellcode-Anmerkung 1*

Die lokalen Gewichte werden im Shared Memory zwischengespeichert. Ein Block berechnet acht lokale Gewichtsänderungen. In Zeile 9 wird der Zwischenspeicher definiert. Laut CUDA Dokumentation soll es auch dynamisch – ohne feste Grenzen – funktionieren. Zeile 10 zeigt dies in der Theorie, was leider nicht funktionierte. Es gab keine Fehlermeldung, sondern schlicht und ergreifend falsche Werte in der Berechnung. Die aktuellen CUDA-Treiber und das aktuelle Toolkit 2.3 steht unter OpenSuSE 10.3 leider nicht zur Verfügung, sodass diese Verifikation offen bleibt.

*Quellcode-Anmerkung 2*

Der Eingabevektor $x_0 \dots x_n$ ergibt sich im sequenziellen Programm immer durch den Nachfolger z. B. $(\overrightarrow{0;0})$, $(\overrightarrow{0;1})$, … $(\overrightarrow{1;0})$, … $(\overrightarrow{7;7})$ (vgl. Abbildung 3-2). Dies ist ein vergleichsweise einfacher und schneller Vorgang. In der parallelen Version wird der Eingabevektor anhand des `bitelement` (vgl. Abschnitt 4.1 und Abbildung 4-4) zurückgerechnet. Dies sind mehrere mathematische Operationen. Die Berechnung ist im Listing 4-3 nicht enthalten. Die definierte maximale Anzahl von $10^{10}$ würde für `bitelement` den Datentyp `long long (int64)` voraussetzen (Zeile 12). Dieser Datentyp existiert auch unter 32 bit Systemen, produziert keine Fehlermeldung, aber falsche Werte. Da am Testsystem keine beliebigen Änderungen vorgenommen werden konnten, bleibt die Gegenprobe auf einem 64 bit Betriebssystem oder 32 bit Betriebssystem mit aktuellen Treibern offen. Aufgrund der Videospeicher-Limitierung (siehe Abschnitt 3.3) ist für den Test `int32` vorerst ausreichend. Allgemein schmälert die Berechnung, zur vergleichsweise einfachen Bestimmung des Nachfolgevektors, die Leistung enorm. Diese Rechnung muss für jeden Sollwert mehrfach durchgeführt werden, da das

optimale Gewicht generell erst nach einigen Zyklen über alle Elemente gefunden wird! Hier könnte es sich lohnen, den Algorithmus nochmals zu überdenken.

*Quellcode-Anmerkung 3*

Im Testprogramm muss der Kernel selbst in einer Schleife aufgerufen werden, da bei entsprechender Größe des Kollisionsfeldes mehr als 65.535 `char`-Elemente existieren (maximale Grid Größe siehe Abschnitt 2.4.3). Zudem findet eine optimiertere Version der parallelen Reduktion Anwendung [Harr07].

## 4.3 Problematik der Programmoptimierung

Optimierungen werden meist aus marktwirtschaftlichen Gründen nicht mehr in dem Umfang wie noch vor 20 Jahren durchgeführt. Oftmals lässt sich die Performance aber dennoch durch geschickte Algorithmen steigern. Im Testprogramm könnte man die Berechnung des Eingabevektors im GPU-Kernel nochmals auf die Probe stellen.

Durch die Vertrautheit der x86-Architektur und das Vorhandensein verschiedener Assembler, inline Assembler bzw. durch den Blick auf den vom Compiler erzeugten Maschinencode können gute Algorithmen durch hardwaretypische Eigenschaften zum Teil zusätzlich optimiert werden.

Ein kleines Beispiel aus der Matrizenmultiplikation soll dies veranschaulichen. Die Matrizenmultiplikation diente als Einstig in die Parallel-Programmierung, ähnlich einem „Hello World“. Für die Genauigkeit der Berechnung ist der Bereich der `float`-Darstellung ausreichend.

Satz: Matrizen werden multipliziert, indem die Elemente einer Zeile der Matrix A mit den Elementen einer Spalte der Matrix B multipliziert werden und im Anschluss die Summe gebildet wird [Papu06, 195f.]. Aus Sicht des Programmierers besteht die Berechnung aus einer dreifach ineinander geschachtelten `for`-Schleife. Das folgende Listing 4-4 zeigt die Matritzenmultiplikation.

**Listing 4-4:** Matrizenmultiplikation

```
01 // Matrizen C = A x B (Matrizen A,    B und C - Datentyp: float*)
02
03 unsigned int i, j, k;                 unsigned int i, j, k;
04
05 // für alle Zeilen (height) in A      // für alle Zeilen (height) A
06 for (i=0; i<hA; i++)                  for (i=0; i<hA; i++)
07   // für alle Spalten (width) in B      // für alle Spalten (width) B
08   for (j=0; j<wB; j++)                 for (j=0; j<wB; j++)
09   {                                    {
10     float sum = 0;                       double sum = 0;
11
12     for (k=0; k<wA; k++)                 for (k=0; k<wA; k++)
13       sum += A[i*wA+k] * B[k*wB+j];        sum += A[i*wA+k] * B[k*wB+j];
14
15     C[i* wB +j] = sum;                   C[i* wB +j] = (float)sum;
16   }                                    }
```

```
01 loc_403520: ; for (k=0; k<wA; k++)
02   fld     dword ptr [ecx]
03   mov     ebp, [esp+3Ch+var_2C]
04   fmul    dword ptr [edi]
05   add     [esp+3Ch+var_2C], eax
06   add     ecx, 10h
07   add     edi, eax
08   fadd    [esp+3Ch+arg_C]
09   fstp    [esp+3Ch+arg_C]
10   fld     [esp+3Ch+arg_C]
11   fld     dword ptr [ecx-0Ch]
12   fmul    dword ptr [ebx]
13   add     ebx, eax
14   faddp   st(1), st
15   fstp    [esp+3Ch+arg_C]
16   fld     [esp+3Ch+arg_C]
17   fld     dword ptr [ecx-8]
18   fmul    dword ptr [ebp+0]
19   mov     ebp, [esp+3Ch+var_28]
20   add     [esp+3Ch+var_28], eax
21   sub     edx, 1
22   faddp   st(1), st
23   fstp    [esp+3Ch+arg_C]
24   fld     [esp+3Ch+arg_C]
25   fld     dword ptr [ecx-4]
26   fmul    dword ptr [ebp+0]
27   faddp   st(1), st
28   fstp    [esp+3Ch+arg_C]
29   jnz     short loc_403520
```

```
loc_40374E: ; for (k=0; k<wA; k++)
  fld     dword ptr [edi]
  mov     ebp, [esp+38h+arg_C]
  fmul    dword ptr [ecx]
  add     [esp+38h+arg_C], eax
  add     ecx, 10h
  add     edi, eax
  faddp   st(1), st
  fld     dword ptr [ecx-0Ch]
  fmul    dword ptr [ebx]
  add     ebx, eax
  faddp   st(1), st
  fld     dword ptr [ecx-8]
  fmul    dword ptr [ebp+0]
  mov     ebp, [esp+38h+var_28]
  add     [esp+38h+var_28], eax
  sub     edx, 1
  faddp   st(1), st
  fld     dword ptr [ecx-4]
  fmul    dword ptr [ebp+0]
  faddp   st(1), st
  jnz     short loc_40374E
```

Links im Listing 4-4 ist ein Beispiel, wie man es vielleicht üblicherweise lösen würde, rechts eine optimiertere Version, die ein wenig Hintergrundwissen voraussetzt. Unter dem C-Code ist jeweils der vom Compiler erzeugte Maschinencode zu sehen.

In diesem Beispiel bringt die Deklaration von `sum` als `double` (Zeile 10, rechte Seite) eine entscheidende Optimierung. Die x86-64 *Floating-point unit (FPU)* arbeitet intern mit double precision. Das heißt es wird ein upcast von `float` zu `double` durchgeführt. Wie im Assembler-Auszug sofort zu sehen ist, spart dies einige Prozessortakte. Die wenigen Prozessortakte mögen zwar angesichts der Rechenleistung moderner CPU banal erscheinen, jedoch summieren diese sich schnell bei einigen tausend

Matrizenelementen zu Sekunden. Bei einzelnen Operationen wird man sich normalerweise zugunsten des Speicherplatzes entscheiden, aber dies ist dem Einzelfall anzupassen.

Bei unvertrauten Technologien, wie ATi Brook+ oder nVIDIA CUDA, ist es schwer auf Anhieb einen optimierten Code zu schreiben und bleibt Experten der Grafikprogrammierung vorbehalten. Ferner steht bei nVIDIA kein Low-Level-Zugriff zur Verfügung und bei ATi wurde die Assembler-Ebene *Close to Metal (CTM)* bereits in den frühen Anfängen komplett durch die Hochsprache Brook+ und das *Compute Abstraction Layer-Interface (CAL-Interface)* ersetzt. Aber auch für die Hochsprachen könnte die Dokumentation und Hardwarebeschreibung von ATi und nVIDIA an vielen Stellen deutlich besser ausfallen! So ist der Technical Brief der GTX 200 GPU [Nvid08] eher eine Marketing-Mappe statt eine Hilfestellung für Programmierer. Bei der Optimierung eines Kernel können Tools wie der Profiler helfen, zeitkritische Programmsequenzen zu analysieren. Inwieweit sich dann, mangels guter Dokumentation, Optimierungsmöglichkeiten finden lassen sei dahingestellt. Um keine Seite zu bevorzugen, ist daher das vorliegende Testprogramm, weder GPU- noch CPU-Seitig, explizit optimiert.

Im Ergebnis kann festgehalten werden, dass die für das Training relevanten Sollwerte bitweise in `char`-Elementen hinterlegt sind. Jeder SM (Streaming Multiprozessor) arbeitet immer mit einem `char`-Element (das heißt 8 Threads je Block) und errechnet eine lokale Gewichtsänderung. Das rechenintensive Problem bei der parallelen Durchführung liegt in der Rückrechnung des zugrundeliegenden Eingabevektors zu einem x-beliebigen `char`-Elementes (vgl. Abbildung 4-4). Die Reihenfolge der Abarbeitung ist nicht deterministisch. Bei der sequenziellen Version bestimmt sich der Nachfolge-Eingabevektor aus dem Vorgänger.

Des Weiteren wurde die parallele Reduktion betrachtet, welche für die parallele Addition der lokalen Gewichtsänderungen innerhalb des SM und über SM-Grenzen hinweg benötigt wird. Darüber hinaus wurden in diesem Kapitel bisher ungelöste Probleme bei der Implementierung aufgeworfen. Zum einen stehen weder aktuelle CUDA-Treiber noch das aktuelle Toolkit 2.3 unter OpenSuSE 10.3 zur Verfügung. Zum anderen produziert der Datentyp `long long` (64 bit Integer) bei hohen Datenmengen unter dem 32 bit System falsche Werte. Zu guter Letzt wurde in diesem Zusammenhang auf den Aspekt der Programmoptimierung eingegangen. Hierbei könnte in Zukunft eine Optimierung durch Verbesserung des Algorithmus vorgenommen werden.

Nach der Ausführung des Programms auf dem Testsystem wird im nachfolgenden Kapitel 5 eine Leistungsbewertung durchgeführt.

# 5 Leistungsbewertung

Nach erfolgreicher Implementierung erfolgt in Kapitel 5 eine Leistungsbewertung der Hardware für einen speziellen Anwendungsfall. Abschnitt 5.1 beschreibt hierzu zunächst die Vorgehensweise der Leistungsbewertung. In Abschnitt 5.2 wird anhand der Matrizenmultiplikation der Frage, ab wann sich der Einsatz der GPU lohnt, nachgegangen. In Abschnitt 5.3 wird die entscheidende Frage der Berechnungsdauer des Neuronalen Netzes auf CPU und GPU beantwortet.

## 5.1 Vorgehensweise

Zur Leistungsbewertung wurden die Berechnungen jeweils auf der CPU und der GPU durchgeführt und die jeweilige Laufzeitdauer ermittelt.

Liegt die zu erwartende Zeitspanne im Millisekunden-Bereich oder reichen Tendenzen, so kann mit den normalen Zeitfunktionen des Computers gearbeitet werden. Werden exaktere Angaben benötigt, so geht dies über den Assembler-Befehl `RDTSC` (Read Time-Stamp Counter). Dieser Zähler wird pro Prozessortakt um eins erhöht und steht seit Pentium II zur Verfügung [Inte09a, 329f.]. Da das Powermanagement moderner Systeme die Taktfrequenz nach belieben anpassen kann, ist die Zeitmessung nicht exakt, bietet aber dennoch die höchstmögliche Genauigkeit auf x86-64 basierenden Systemen.

Bei der Programmierung wird man unter gewöhnlichen Umständen eine Ebene höher ansetzen und das *Advanced Programming Interface (API)* des jeweiligen Betriebssystems nutzen. Microsoft Windows stellt mit `QueryPerformanceCounter` und `QueryPerformanceFrequency` [Msdn08b] bzw. Linux mit `clock_gettime` Zugriff auf den Zähler zur Verfügung.

Auch das CUDA-SDK beinhaltet Funktionen zur Zeitmessung: `cutCreateTimer`, `cutStartTimer`, `cutStopTimer`, `cutGetTimerValue`, `cutResetTimer`, `cutDeleteTimer`. Intern werden diese Funktionen wahrscheinlich auf die API des jeweiligen Betriebssystems zurückgreifen. Aufgrund der Plattformunabhängigkeit bei der Programmierung und der recht einfachen Handhabung wurde zur Bestimmung der Laufzeitdauer die CUDA-Funktionalität genutzt.

## 5.2 Matrizenmultiplikation

In Abschnitt 3.3.3 wurde die These aufgestellt, dass es nicht sinnvoll ist einzelne Berechnungen auf der GPU auszuführen. Des Weiteren wurde behauptet, dass es schon

mindestens eine Millionen gleichartiger Berechnungen benötigt, oder teilweise entscheidend ist, ob Daten im Grafikspeicher mehrfach verwendet werden können.

Diese Behauptungen können zunächst anhand einer einfachen Matrizenmultiplikation bestätigt werden. Eine Matrizenmultiplikation ist sequenziell äußerst einfach umzusetzen (Listing 4-3) und im Umfeld der Parallel-Programmierung sehr gut als Einstieg geeignet.

Im ersten Fall wurden zwei $8\times8$ Matrizen im Host-Speicher angelegt, mit Zufallswerten gefüllt. In Summe sind dies zweimal 64 Elemente je 4 Byte: Es musste also 512 Byte von Host zu Device transferiert werden und 256 Byte (entstandene $m\times n$ Matrix nach der Multiplikation) von Device zu Host transferiert werden. Zur Berechnung waren 512 Iterationen notwendig. Die Initialisierung der CUDA-Runtime wird automatisch vorgenommen, sobald eine Runtime-Funktion erstmalig aufgerufen wird. Werden mehrere Kernel nacheinander ausgeführt, so ist dies für gewöhnlich ein einmaliger Vorgang. Da einzelne Berechnungen verglichen werden, ist dieser Anteil jedes Mal vorhanden. Anhand der $8\times8$ Matrix ist ersichtlich, dass sich einzelne Berechnungen nicht effizient auf der GPU durchführen lassen. Wie indirekt aus Abschnitt 2.4.3 entnehmbar, ist die Mindestmenge an Daten die ein SM abarbeitet 32 Threads (1 Warp). Die meisten Prozessoren sind daher ilde.

Im zweiten Fall der $64\times64$ Matrix könnte unter Umständen von der GPU profitiert werden, sofern die transferierten Daten mehrfach genutzt werden. Die eigentliche Berechnung wurde von der GPU rund zehnmal schneller durchgeführt, da hier die Prozessoren bereits gut ausgelastet sind.

Im dritten Fall profitiert man bei über 134 Millionen Iterationen deutlich von der Leistungsfähigkeit.

Alle drei Fälle sind grafisch in Abbildung 5-1 zusammenfassend dargestellt. Auf Grund der Tatsache, dass der Wertebereich von 0,002 ms bis 395 ms sehr groß ist, wurde das Balkendiagramm logarithmisch skaliert und für jede Matrix ein eigener Maßstab gewählt. Die Initialisierung wurde als letztes aufgeschlagen, sodass eine bessere visuelle Gegenüberstellung erfolgen kann.

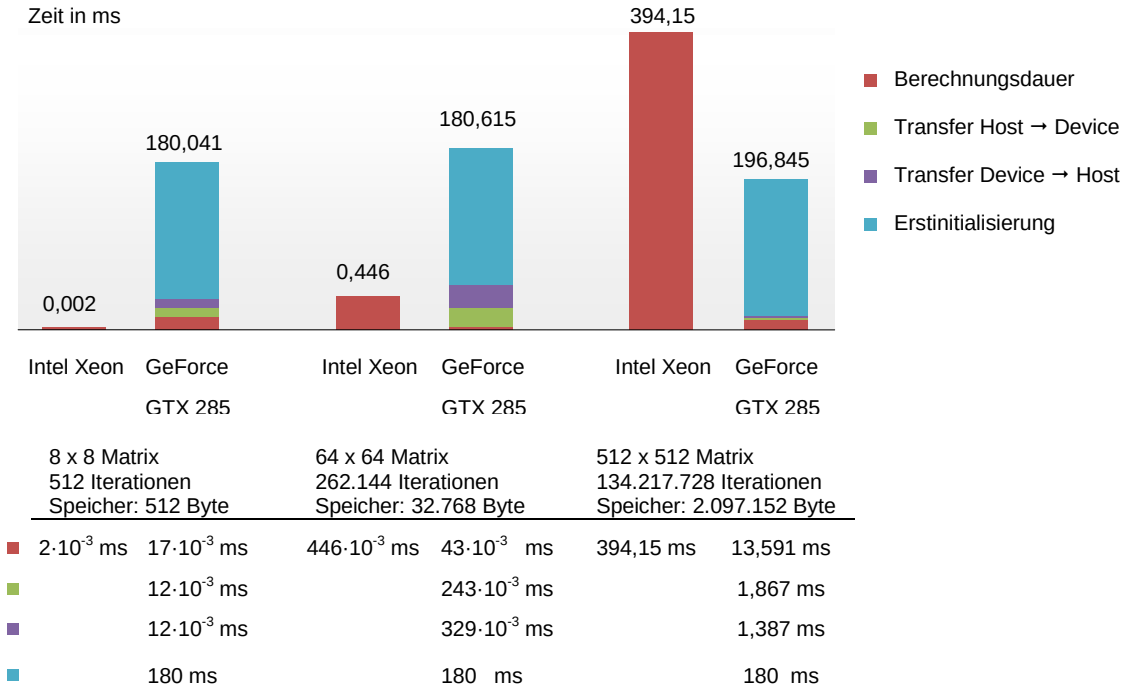



**Abbildung 5-1:** Berechnungsdauer Matritzenmultiplikation

Nachfolgend wird die Berechnungsdauer des Neuronalen Netzes untersucht.

## 5.3 Berechnungsdauer des Neuronalen Netzes

Als Testgrundlage wurde das Kollisionsfeld mit $9^9$ Elementen gefüllt. Für die bitweise Speicherung sind hierfür 48.427.562 `char`-Elemente, je 1 Byte nötig. Das sind zirka 46,18 MB.

Das Training mittels GPU setzt voraus, dass die für die Berechnung nötigen Strukturen in den Grafikspeicher kopiert werden. Die Dauer der Erstinitialisierung der CUDA-Runtime und der Transfer des Kollisionsfeldes von Host zu Device ist ein nahezu konstanter Wert, welcher als Bestandteil der Berechnungsdauer jeweils aufgeschlagen wurde. Der Transfer des summierten Gewichtes (4 Bytes) von Device zu Host kann nahezu vernachlässigt werden. Die folgende Abbildung 5-2 listet die Anteile im Einzelnen auf.

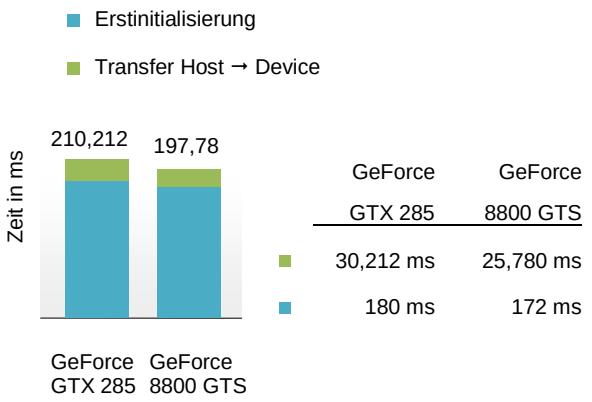



**Abbildung 5-2:** Initialisisierung der CUDA-Runtime und Transferzeit

Interessanterweise stellte sich für diesen Fall heraus, dass die ältere GeForce 8800 GTS, wenn auch nur marginal, schneller ist. Diese 6 % Leistungsunterschied vermag auch Tabelle 3-4 nicht recht erklären. Es ist zu vermuten, dass die Erstinitialisierung der deutlich mehr verbauten Komponenten zusätzliche Zeit in Anspruch nimmt. Aber auch der Transfer von Host zu Device war bei der älteren GeForce 8800 GTS schneller. Hier wird die enthaltene Allokation des Speichers den entscheidenden Ausschlag geben. Bei der recht kleinen Datenmenge kann die höhere Bandbreite der GeForce GTX 285 letztendlich nicht den entstandenen Zeitverlust ausgleichen. Betrachtet unter dem Aspekt, dass derzeit kein Spiel 1 GB Grafikspeicher fordert, wurde in diversen Gaming-Foren schon rege diskutiert, dass Grafikkarten des gleichen Types, aber mit weniger Speicher oftmals schneller sind. Für GPGPU sollte man sich dennoch zugunsten des Speichers entscheiden.

Demnach ist zu erwarten, dass die reine Rechenkarte nVIDA Tesla C1060 zur GeForce GTX 285 identische Werte bezüglich Initialisierung und Netto-Transfer liefern wird. Die Zeitdauer für Allokation des Speichers wird jedoch geringfügig höher ausfallen.

Nachdem die Hardware initialisiert und die nötigen Strukturen in den Grafikspeicher kopiert wurden, wurde das Neuronale Netz je $3\times$ mit jeweils unterschiedlicher Lernrate trainiert. Eine kleinere Lernrate bedeutet eine kleinere Gewichtserhöhung und ein besseres Lernergebnis, jedoch auch eine längere Lernzeit. Zyklen bedeutet wie oft das gesamte Kollisionsfeld ($9^9$ Sollwerte) durchlaufen wurde, um das Gewicht zu ermitteln.

Bis zu einem gewissen Punkt kann durch Verkleinerung der Lernrate das Lernergebnis stetig verbessert werden. Das genaue Ausloten der Lernrate ist an dieser Stelle nicht weiter von Bedeutung. Bei der parallelen Summation ergeben sich zur sequenziellen gewisse Abweichungen. Das Lernergebnis ist im Bereich der `float`-Genauigkeit jedoch identisch.

Die Lernzeit wird in Abbildung 5-3 grafisch gegenüber gestellt.

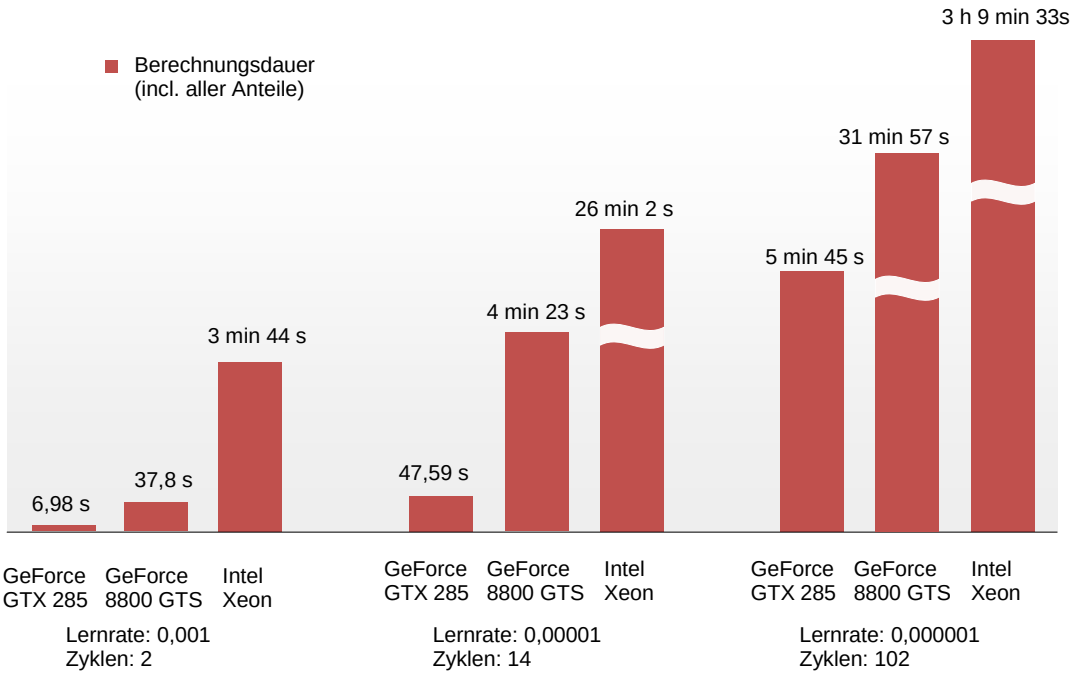



**Abbildung 5-3:** Trainingsdauer des Neuronalen Netzes

Der Kernel des Neuronalen Netzes besteht aus recht viel Overhead und wenig Datenoperationen (siehe Listing 4-3). Da beim $512 \times 512$ Matrixtest eine ähnliche Konstellation vorlag, lag die Erwartung ebenfalls bei einer zirka 29-fach schnelleren Abarbeitung unter Zuhilfenahme der GeForce GTX 285 gegenüber der Intel Xeon CPU. In der Praxis ergaben sich die in Tabelle 5-1 gegenübergestellten Werte, welche sich auch aus Abbildung 5-3 ablesen lassen.

**Tabelle 5-1:** Leistungssteigerung GPU – CPU

| | GeForce 8800 GTS | Intel Xeon |
|---|---|---|
| GeForce GTX 285 | 5,5x | 33x |

| | Intel Xeon |
|---|---|
| GeForce 8800 GTS | 5,9x |

Die Erwartung wurde sogar etwas übertroffen, was auf eine gute Auslastung der Prozessoren und hohen Parallelitätsgrad zurückzuführen ist. Die Berechnung konnte mit Hilfe der GeForce GTX 285 rund $33\times$ schneller durchgeführt werden als mit einer Intel Xeon CPU. Für die Praxis ist dies bereits ein gutes Ergebnis. Durch Erweiterung der Kenntnisse in der GPU Programmierung und Änderungen am Kernel ist eine um Faktor $50\times$ bis $60\times$ schnellere Berechnung ein durchaus realisierbares Ziel.

Richtig interessant wird der Leistungszuwachs, wenn selbst die GPU über Nacht rechnen muss; was der Fall sein wird, wenn das Neuronale Netz aus mehreren Knoten und zirka $10^{10}$ Trainingswerten besteht.

Der Leistungsunterschied zwischen GeForce GTX 285 und der älteren GeForce 8800 GTS kann als nahezu konstant betrachtet werden, da dieser hauptsächlich auf die Taktung und Prozessorenanzahl zurückzuführen ist.

Seitens der CPU ist eine theoretisch bis zu 8× schnellere Berechnung möglich, wenn durch „threading“ alle 8 Cores der Xeon CPU ausgelastet werden. Somit würde sich der Vorteil der GeForce GTX 285 aktuell auf „nur noch“ 4⅛× verkleinern. Die alte GeForce 8800 GTS wäre langsamer. Dem gegenüber sei nochmals erwähnt, dass bis zu vier Grafikkarten pro System verbaut werden können. Andere Anmerkungen zur Optimierung wurden bereits in Abschnitt 4.3 angesprochen.

In diesem Kapitel erfolgte der Leistungsvergleich zwischen der nVIDIA GeForce 8800 GTS, der nVIDIA GeForce GTX 285 und der Intel Xeon CPU. Im Ergebnis der Leistungsbewertung konnte festgestellt werden, dass die Trainingsdauer des Neuronalen Netzknoten unter Zuhilfenahme der GPU deutlich gesenkt werden konnte. Somit kann bestätigt werden, dass die auf Basis der Marktuntersuchung ausgewählte Grafikkarte für die Berechnung des Neuronalen Netzes geeignet und folglich effizienter und effektiver als die bisher verwendete CPU ist.

# 6 Schlussbetrachtungen

Das folgende Kapitel gibt eine Zusammenfassung der Ergebnisse (Abschnitt 6.1) sowie einen Ausblick auf zukünftige Entwicklungen (Abschnitt 6.2).

## 6.1 Zusammenfassung der Ergebnisse

Im Rahmen dieser Diplomarbeit wurden die technischen und theoretischen Grundlagen Neuronaler Netze und Grafikkarten sowie deren Funktionsweise erörtert, wodurch die vorliegende Arbeit zu einem besseren Verständnis rund um das Thema Grafikkarten und Neuronale Netze beiträgt.

Darüber hinaus wurde ein Programm entwickelt, auf dessen Grundlage eine Weiterentwicklung und Erweiterung für andere Neuronale Netz Probleme möglich ist.

Des Weiteren gibt die Arbeit eine Hilfestellung für die Abteilung MEA1, welche Grafikhardware für die mathematischen Berechnungen geeignet ist und in Zukunft angeschafft werden könnte. Zu diesem Zweck wurde ein aktueller Marktüberblick über handelsübliche Grafikkarten im PC-Umfeld sowie Grafikboards im VME- und cPCI-Bereich mit entsprechender Spezifikation erstellt. Darauf aufbauend wurden die aktuell am Markt erhältlichen Grafikkarten auf ihre Eignung hin evaluiert, um so eine geeignete Grafikkarte für die mathematischen Berechnungen zu ermitteln. Im Rahmen dieser Evaluierung wurde sich für die nVIDIA GeForce GTX 285 entschieden.

Danach wurde in Koordination mit der Fachabteilung ein Neuronales Netz zum Test für die Leistungsbewertung der ausgewählten GeForce GTX 285 entworfen. Aufbauend auf diesem Entwurf wurde die Implementierung des Testprogramms vorgenommen. Um festzustellen, ob ein signifikanter Leistungsunterschied vorliegt und um eine Messung der jeweiligen Berechnungsdauer vorzunehmen, wurde das Neuronale Netz sowohl auf die CPU als auch auf die GPU der GeForce GTX 285 portiert.

Im Anschluss an die Implementierung erfolgte eine Leistungsbewertung, bei der die Eignung der GeForce GTX 285 für die Berechnung des Neuronalen Netzes untersucht wurde. Hierbei konnte gezeigt werden, dass die Berechnung eines Neuronalen Netzes deutlich von der GPU profitieren kann. Es konnte eine Verringerung der Berechnungsdauer um Faktor 33 festgestellt werden. Aber auch für andere, auf SIMD abbildbare, rechenintensiven Probleme kann eine GPU in Betracht gezogen werden.

Keine andere Technik bietet derzeit gleiche Leistung pro Watt oder Gflop pro Euro. GPGPU ist für alle Technikversierten, Forschung und Entwicklung die aktuell viel Leistung zu günstigen Preisen wünschen und nicht auf die kommende CPU Generationen warten können höchst interessant.

Wer momentan noch etwas warten kann, sollte erst demnächst umsteigen, wenn DirectX 11 kompatible Hardware verfügbar ist. Damit erwirbt man Komponenten, die zumindest von der Technologie die nächsten Jahre bestand hat. ATi fertigt bereits erste DirectX 11 GPU. Die neue FireStream- und Tesla-Serie wird jedoch noch einige Entwicklungszeit beanspruchen.

Folglich können durch die Verwendung einer handelsüblichen Grafikkarte die Berechnungsdauer des Neuronalen Netzes nicht nur signifikant verringert werden, sondern auch Effizienzmehrwerte und Effektivitätsmehrwerte für den Bereich MEA1 gewonnen werden.

## 6.2 Ausblick

Die Festlegung auf nVIDIA CUDA oder ATi Brook+ ist für Spezial-Anwendungen kein großes Problem, jedoch für Standard-Software. Hier werden es sich die Software-Häuser nicht leisten können mit einer Technologie nur rund ein Drittel des Marktes abzudecken. Eine Portierung zwischen den beiden Sprachen wird einer Neuprorgammierung gleich kommen. Diese Lücke wird durch DirectX 11 und *Open Computing Language (OpenCL)* geschlossen.

Wie schon bei OpenGL wird wohl jeder Hersteller seine spezifische OpenCL Implementierung veröffentlichen. Auch wenn OpenCL plattformunabhängig ist, kommt es vorzugsweise unter Linux und MacOS zum Einsatz. Da bei OpenCL die Hardware nicht bekannt sein muss, wird ein OpenCL-Programm erst zur Laufzeit übersetzt (vgl. Abbildung 2-22). In der nVIDIA OpenCL-Implementierung wird ebenso PTX-Code erzeugt.

Unter Microsoft Windows wird erfahrungsgemäß auf den De-facto-Standard DirectX zurückgegriffen. Microsoft hat mit DirectX 11 den *Compute Shader* für Berechnungen spezifiziert. Dies ist sozusagen ein weiterer Operationsmodus des Shaders (siehe Abschnitt 2.4.1). Hier könnten demnächst auch Hersteller mit DirectX 11 kompatibler Hardware, die aber kein eigenes SDK anbieten, für GPGPU interessant werden.

Inwieweit die Programmiermodelle nVIDIA CUDA und ATi Brook+ ihre Daseinsberechtigung auch in Zukunft rechtfertigen ist ungewiss.

Aber auch seitens der CPU ist eine Abwendung von der von Neumann Architektur, wo sequenzielle Programme mit hoher Geschwindigkeit abgearbeitet werden, zu erkennen. Laut Medien plant AMD eine „Fusion“ von CPU und GPU, Intel setzt mit „Larrabee“ auf ein Array von x86-64 Prozessoren und bezeichnet CUDA als eine interessante Fußnote in der Geschichte [Hard08].

Abbildung 6-1 fasst die aktuellen Presseberichte und Marketingmeldungen[3] der drei Marktführer zusammen.

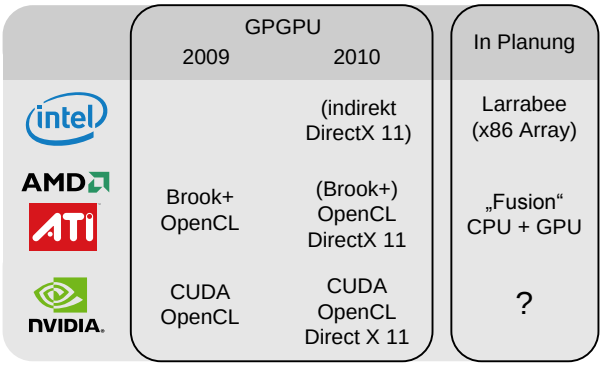



**Abbildung 6-1:** Prozessorgeflüster

Beim genaueren Betrachten sind die Vorhaben lediglich „alter Wein in neuen Schläuchen“. Die Integration der GPU in die CPU ist ein IGP an anderer Stelle und Larrabee ein größeres Upgrade vorhandener Mehrkern-CPU. Auch wenn exklusiv angekündigt, wird sich Erfahrungsgemäß die Produktpalette von AMD und Intel ähneln.

Im Gegensatz zu CPGPU hat die Programmierung der CPU den Vorteil, dass die Einarbeitung in ein neues Programmiermodell entfallen kann. Mit dem Intel Parallel Studio oder der kommenden Version des Microsoft Visual Studio 2010 wird Parallel-Programmierung für x86-64 kompatible Prozessoren noch einfacher, sodass bei entsprechender Leistungssteigerung die an die GPU verloren gegangenen Prozente zurückkehren könnten. Zudem ist die MIMD-Architektur (siehe Abschnitt 2.4.3) der CPU für die meisten Problemstellungen günstiger.

Noch offen ist die weitere Strategie von nVIDIA. Auch wenn es im Spielemarkt noch genügend Potential gibt, schrumpfte nach aktuellen Prognosen der GPU Marktanteil im Q4/2009 zu Gunsten von ATi und Intel.

Wirkliche 3D-Technik – nicht nur 3D Bilder auf 2D Bildschirmen, der Netbook und (Mobilfunk-)Embedded Markt könnten in Zukunft weitere bedeutende Rollen spielen. Die zukünftige Marktentwicklung bleibt also spannend.

[3] Stand vom 01.11.2009

# Anhang

## A Erklärung zur Diplomarbeit

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich alle verwendeten Quellen, auch Internetquellen, ordnungsgemäß angegeben habe.

Manching und Mittweida, den 20.06.2010 _____
(Vorname, Nachname)

## B Anleitung: Einrichten von Eclipse für CUDA unter Linux

Nach der Installation des nVIDIA CUDA-Treibers, des CUDA-Toolkit und des CUDA-SDK soll das Testprogramm „CUDABench“ unter Eclipse eingerichtet werden.

Es werden die Standard-Pfade angenommen:
Toolkit: /usr/local/cuda/ SDK: (userhome) ~/NVIDIA_CUDA_SDK/

1. Compilierung der benötigten Bibliotheken (Libraries)
   - cd ~/NVIDIA_CUDA_SDK/common → make

   Die Demo-Projekte des SDK sollten jetzt über die Konsole kompilierbar sein.

2. Nutzung von CUDABench unter Eclipse
   - Ordner CUDABench/ von CD nach ~/NVIDIA_CUDA_SDK/projects/ kopieren

   In Eclipse:
   - New→C++ Project
     - Project Name: CUDABench
     - "Use default location" abwählen
     - Speicherort: ~/NVIDIA_CUDA_SDK/projects/CUDABench
     - Project type: Executable→Empty Project
   - Next→
     - "Debug" abwählen
     - "Release" wählen→Advanced Settings→"C/C++ Build"
       - Tab "Builder Settings"→"Generate Makefiles" abwählen
       - Build Directory: "${workspace_loc:/CUDABench}"
       - Tab "Behaviour"→Build (incremental build)→"all" entfernen
     - "C/C++ Build"→Environment
       - Add→Name: "PATH" Value: "/usr/local/cuda/bin"→OK
       - Bugfix: Select PATH→Edit→Value: ";/usr/local/cuda/bin" zu ":/usr/local/cuda/bin" ändern
   - OK→Finish

3. Syntax Highlighting für CUDA *.cu Dateien

   - Window→C/C++→File Types→New→Eingabe *.cu, C++ Source File

   Das Syntax Highlighting funktioniert nur für den C++ Teil innerhalb der *.cu Dateien. Die CUDA-Spezifische Syntax beherrscht das CDT-Plugin nicht.

4. Anmerkungen

   - Emulationsmodus: Im Makefile folgendes eintragen bzw. auskommentieren

     ```
     # Rules and targets #
     emu=1
     include ../../common/common.mk
     ```

   - Kommt es beim Ausführen der Executable zu Fehlermeldungen bitte Prüfen:

     - Benutzer muss in Gruppe "Video" sein (oder root-Rechte besitzen)
     - LD_LIBRARY_PATH=/usr/local/cuda/lib
       export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib
       oder in Eclipse einstellen

## C Inhaltsverzeichnis CD

Auf der beiliegenden CD befindet sich das Testprogramm „CUDABench“, Excel-Tabellen über den Marktüberblick und die Auswertung der Performanceentwicklung, sowie einige Literatur-Referenzen.

| | | |
|---|---|---|
| CD-ROM:\ | | |
| file_id.diz | | |
| pgp_key.asc | | |
| Diplomarbeit.pdf | | |
| Anhang\Anhang D | | |
| PC Grafikkarten Marktueberblick.xls | | |
| VME cPCI Grafikmodule Marktueberblick.xls | | |
| Anhang\Anhang E | | |
| Performanceentwicklung.xls | | |
| Referenzen | | |
| Anal95.pdf | Hdmi06.pdf | Nvid09a.pdf |
| Ati09.pdf | Ieee08.pdf | Nvid09b.pdf |
| BAD+01.pdf | Inte09a.pdf | Nvid09c.pdf |
| BAD+01_ppt.pdf | Inte09b.pdf | Papu06.pdf |
| Bert09a.pdf | Kauf06.pdf | Pedd07.pdf |
| Bert09b.pdf | KuDo06.pdf | Pedd09.xls |
| Bish06.pdf | Kuhn03.pdf | PWA+08.pdf |
| Ddwg99.pdf | LNOM08.pdf | Qimo07.pdf |
| Epfl09.pdf | Maxp09.pdf | Rose58.pdf |
| FeKi03.pdf | McPi43.pdf | RuHW86.pdf |
| FiSt09.pdf | MoJB07.pdf | ScPW09.pdf |
| Fuji08.pdf | Moor65.pdf | Shal08.pdf |
| Gaud06.pdf | Moor75.pdf | SiIi04.pdf |
| Gepp05.pdf | Moto77.pdf | Ucsf04.jpg |
| Hard08.pdf | Msdn08a.pdf | WiBL08.pdf |
| Harr07.pdf | Msdn08b.pdf | Wild60.pdf |
| Hdcp06.pdf | Nvid08.pdf | YaBZ02.pdf |
| CUDABench | | |
| main.cu | matrixtests.cu | CUDABench.aps |
| Makefile | matrixtests.h | CUDABench.ncb |
| collisionfield.cu | devices.cu | CUDABench.rc |
| collisionfield.h | devices.h | CUDABench.sln |
| neuronalnet.cu | resource.h | CUDABench.vcproj |
| neuronalnet.h | CUDA.Rules | vc90.pdb |

## D GPU-Marktüberblick

Aufgrund des Datenumfangs befindet sich der Marktüberblick als Anlage auf CD.

```
CD-ROM:\Anhang\Anhang D
        PC Grafikkarten Marktueberblick.xls
        VME cPCI Grafikmodule Marktueberblick.xls
```

## E GPU-Performanceentwicklung

Aufgrund des Datenumfangs befinden sich die Daten der Performanceentwicklung als Anlage auf CD.

```
CD-ROM:\Anhang\Anhang E
        GPU-Performanceentwicklung.xls
```

# Literaturverzeichnis


[AlBG07] *Alparslan, A..; Bächstädt, K.-H.; Geldermann, A.*: Systeme und Kriterien des Finanzratings. In: *Achleitner, A.-K.; Everling, O.; Niggemann, K. A. (Hrsg.)*: Finanzrating - Gestaltungsmöglichkeiten zur Verbesserung der Bonität. Gabler, Wiesbaden 2007.

[Anal95] *Analog Devices*: Specification True-Color Video RAM DAC ADV7160 vom 25.10.1995. http://www.analog.com, letzter Abruf am 20.06.2009.

[Ati09] *ATi*: R700-Family Instruction Set Architecture Revision 1.0 vom 31.03.2009.

[BAD+01] *Barlage, D.; Arghavani, R.; Dewey, G.; Doczy, M.; Doyle B.; Kavalieros, J.; Murthy, A; Roberds, B; Stokley, P.; Chau, R.*: High-Frequency Response of 100nm Integrated CMOS Transistors with High-K Gate Dielectrics. In: Components Research Logic Technology Development Intel Corporation. http://www.intel.com, letzter Abruf am 26.05.2009.

[Bert09a] *Bertuch, M.*: Klötzchenwelten. In: Magazin für Computertechnik (c't) Nr. 04 (2009), S. 182-187.

[Bert09b] *Bertuch, M.*: Parallel-Werzeuge. In: Magazin für Computertechnik (c't) Nr. 11 (2009), S. 142-147.

[Bish06] *Bishop, C.*: Pattern Recognition and Machine Learning., Springer, New York 2006.

[Ddwg99] *Digital Display Working Group*: Digital Visual Interface DVI Revision 1.0 vom 02.04.1999. http://www.ddwg.org, letzter Abruf am 17.06.2009.

[Epfl09] *EPFL*: 112-bit prime ECDLP solved. http://lacal.epfl.ch, letzter Abruf am 05.08.2009.

[FeKi03] *Fernando, R.; Kilgard, M. J.*: The Cg Tutorial. Addison-Wesley, Boston 2003.

[FiSt09] *Fischer, M.; Stiller, A.*: Feurige Tesla-Strömungen. In: Magazin für Computertechnik (c't) Nr. 11 (2009), S. 148-149.

[Fuji08] *Fujimoto, N.*: Faster Matrix-Vector Multiplication on GeForce 8800 GTX (2008).

[Gaud06] *Gaudlitz, R.*: Neuronale Netze – Vorlesungsscript Kapitel 1 vom 18.12.2006.

[Gepp05] *Geppert, L.*: Power to the Molecules - A crossbar latch supplies the missing piece for a nanosize alternative to the transistor. In: IEEE Spectrum (2005), S. 18-19.

[Hard08] *Hardwidge, B.*: CUDA will be just a "footnote" in computing history. http://www.custompc.co.uk, letzter Abruf am 17.08.2009.

[Harr07] *Harris, M.*: Optimizing Parallel Reduction in CUDA vom 14.11.2007.

[Hdcp06] *Digital Content Protection LLC*: High-bandwidth Digital Content Protection System Revision 1.3 vom 21.12.2006. http://www.digital-cp.com, letzter Abruf am 17.06.2009.

[Hdmi06] *HDMI Licensing*: High-Definition Multimedia Interface Specification Version 1.3a vom 02.04.1999. http://www.hdmi.org, letzter Abruf am 17.06.2009.

[Hebb02] *Hebb, D. O.*: The Organization of Behavior - A Neuropsychological Theory. Nachdruck, Lawrence Erlbaum Assoc Inc, New York 2002.

[Ieee08] *IEEE Computer Society*: IEEE Standard for Floating-Point Arithmetic. In: IEEE Std 754 (2008).

[Inte09a] *Intel*: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B vom 27.09.2009.

[Inte09b] *Intel*: Intel P45 Express Chipset Block-Diagram. http://www.intel.com, letzter Abruf am 07.01.2010.

[Kauf06] *Kaufmann, J.*: AMD kauft ATi. http://www.zdnet.de, letzter Abruf am 06.12.2009.

[KuDo06] *Kurzak1, J.; Dongarra, J.*: Implementation of a Mixed-Precision in Solving Systems of Linear Equations on the CELL Processor vom 26.11.2006.

[Kuhn03] *Kuhn, M.*: Compromising emanations - eavesdropping risks of computer displays (2003). http://www.cl.cam.ac.uk, letzter Abruf am 17.06.2009.

[LNOM08] *Lindholm, E.; Nickolls, J.; Oberman, S.; Montrym, J.*: NVIDIA TESLA – A Unified Graphics and Computing Architecture. In: IEEE Micro Hot Chips Vol. 19 (2008), S. 39-55.

[Maxp09] *Max-Planck-Institut für Quantenoptik*: Der Quantensprung zum Verständnis der Natur. http://www.mpq.mpg.de, letzter Abruf am 01.07.2009

[McPi43] *McCulloch, W. S.; Pitts, W.*: A logical calculus of the ideas immanent in nervous activity. In: Bulletin of Mathematical Biology Nr. 5.4 (1943), S. 115-133.

[Mess94] *Messmer, H.-P.*: PC-Hardwarebuch. 2. erw. Aufl., Addison-Wesley, Bonn 1994.

[Mind02] *MindCreators*: Neuron Basics. http://www.mindcreators.com/NeuronBasics.htm, letzter Abruf am 15.10.2009.

[MiPa69] *Minsky, M.; Papert, S.*: Perceptrons - An Introduction to Computational Geometry. The MIT Press, 1969.

[MoJB07] *Mohamed, T.; Jullien, G.; Badawy, W.*: Crossbar Latch-based Combinational and Sequential Logic for nano FPGA. In: IEEE International Symposium on Nanoscale Architecture (2007), S. 117-122.

[Moor65] *Moore, G. E.*: Cramming more components onto integrated circuits. In: Electronics Vol. 38 Nr. 8 (1965).

[Moor75] *Moore, G. E.*: Progress in Digital Integrated Electronics. In: International Electron Devices Meeting, IEEE (1975), S. 11-13.

[Moto77] *Motorola*: Specification MC6845 CRT Controller (1977). http://www.motorola.com, letzter Abruf am 20.06.2009.

[Msdn08a] *Microsoft Developer Network – MSDN Library 2008*: C/C++ Bitfields. http://msdn.microsoft.com, letzter Abruf am 27.10.2009.

[Msdn08b] *Microsoft Developer Network – MSDN Library 2008*: Zeitfunktionen. http://msdn.microsoft.com, letzter Abruf am 02.11.2009.

[Nvid08] *nVIDIA*: Technical Brief – NVIDIA GeForce GTX 200 GPU Architectural Overview vom 17.06.2008.

[Nvid09a] *nVIDIA*: NVIDIA CUDA Programming Guide Version 2.3 vom 01.07.2009.

[Nvid09b] *nVIDIA*: NVIDIA CUDA C Programming Best Practices Guide Version 2.3 vom 01.07.2009.

[Nvid09c] *nVIDIA*: High Performance Computing with CUDA – 2009 Users Group Conference San Diego vom 15.06.2009.

[Papu06] *Papula, L.*: Mathematische Formelsammlung für Ingenieure und Naturwissenschaftler. 9. erw. Aufl., Vieweg, Wiesbaden 2006.

[Pedd07] *Jon Peddie Research*: Market Watch Fourth Quarter 2007 Version 1.1 Graphics Semiconductor shipments and market activity vom 01.02.2008. http://jonpeddie.com/press-releases, letzter Abruf am 27.05.2009.

[Pedd09] *Jon Peddie Research*: Intel and Nvidia Come Roaring Back in Q1'09 According to Jon Peddie Research. http://jonpeddie.com/press-releases, letzter Abruf am 27.05.2009.

[PWA+08] *Patel, N.; Wakharkar V; Agrahram, S.; Deshpande N.; Pang, M.; Tanikella, R.; Manepalli, R.; Stover, P.; Jackson, J.; Mahajan, R.; Tiwari, P.*: Flip-Chip Packaging Technology for Enabling 45nm Products. In: Intel Technology Journal Vol. 12 vom 17.06.2008. http://www.intel.com, letzter Abruf am 26.05.2009.

[Qimo07] *Qimonda*: GDDR5 – White Paper vom 24.08.2007. http://www.qimonda.com, letzter Abruf am 15.06.2009.

[Rose58] *Rosenblatt, F.*: The perceptron - a probabilistic model for information storage and organization in the brain. In: Psychological Reviews Nr. 65 (1958), S. 386-408.

[RuHW86] *Rumelhart, D. E.; Hinton, G. E.; Williams, R. J.*: Learning representations by back-propagating errors. In: Letters to Nature, Vol. 323 (1986), S. 533-536.

[ScPW09] *Schroeder, B.; Pinheiro, E.; Weber, W.-D.*: DRAM Errors in the Wild – A Large-Scale Field Study vom 07.05.2009.

[Shal08] *Shalev-Shwartz, S.*: Perceptron Algorithm. In: *Kao, M.-Y. (Hrsg.)*: Encyclopedia of Algorithms. Springer, New York 2008.

[Sili04] *Silicon Image*: Digital Visual Interface & TMDS Extensions White Paper vom 13.10.2004. http://www.siliconimage.com, letzter Abruf am 20.06.2009.

[Ucsf04] *University of California San Francisco (UCSF)*: Visualization of Virus Capsids. http://www.cgl.ucsf.edu/Research/virus, letzter Abruf am 26.05.2009.

[Urch02] *Urchs, M.*: Maschine, Körper, Geist - Eine Einführung in die Kognitionswissenschaft. Klostermann, Göttingen 2002.

[WiBL08] *Windeck, C.; Briegleb, V.; Labs, L.*: Nvidia-Grafikchips mit Lebensdauerproblemen. http://www.heise.de, letzter Abruf am 02.07.2009.

[Wild60] *Widrow, B.*: An Adaptive "ADALINE" Neuron using chemical "MEMISTORS". Technical Report 1553-2, 1960.

[YaBZ02] *Yanling, Z.; Bimin, D.; Zhanrong, W.*: Analysis and Study of Perceptron to Solve XOR Problem. In: Proceedings of the 2nd Intemational Workshop on Autonomous Decentralized System (2002), S. 168-173.

[Zell03] *Zell, A.*: Simulation neuronaler Netze. Oldenburg Verlag, München 2003.